سلسلة تعلم ببساطة

SIMPLY AVR

From Blinking LED to RTOS

تعلم أساسيات الأنظمة المدمجة

من تشغيل دايود ضوئي إلى أنظمة الوقت الحقيقي





عبد الله علي عبد الله







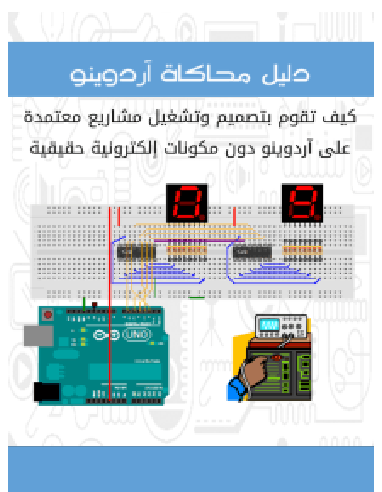


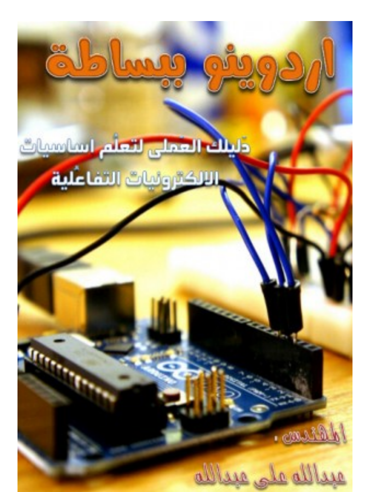
سلسلة ” تعلم ببساطة “

تهدف سلسلة كتب” تعلم ببساطة “ إلى توفير علوم الإلكترونيات الحديثة باللغة العربية بصورة مجانية ومفتوحة المصدر مع الحفاظ على المبدأ الأساسي “**البساطة**” في شرح المعلومات، جميع الكتب موجهة للعامة من هواة الإلكترونيات و طلبة الكليات الهندسة.









رخصة الكتاب

يمكنك تحميل السلسلة من الموقع http://simplyarduino.com/?page_id=889

للتواصل مع المؤلف

abdallah.ali.abdallah.elmasry@gmail.com

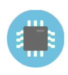
الغلاف مُصمم بواسطة نور الدين – سوريا

http://fb.com/NourHamda.Portfolio

nouraldean.sy@gmail.com

عبدالله علي عبدالله

1437 هـ الموافق 2015 م.



إهداء

أبي و أمي ... لولاكما ما تعلمت حرفاً ..

أساتذة هندسة الحاسبات بجامعة حلوان وأخص منهم،

أ.د. محمد العدوي

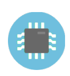
شكراً لأنك أحببت العربية، شكراً جزيلاً على كل كتبك الرائعة، فهي ذخر للأمة

أ.د. علاء حمدي

شكراً على تبسيطك للعلم، يعلم الله كم البهجة الذي أدخلته على قلوب الطلبة :))

د. أحمد يوسف

شكراً على إخلاصك، تحملك للطلبة والإصرار على إيصال العلم، شكراً



شكراً لكل من شارك

م. أحمد أُسامة – شكراً للمساهمة بفصل الـ UART

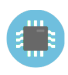
م. يحيى طويل – شكراً على نصائحك ومقالات "عتاديات" الرائعة

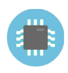
شكراً لكل من ساهم بمراجعة الكتاب

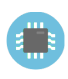
م. حمدي سلطان، م. سعيد الشايب، أحمد م. أبو زيد، إسلام الليثي، محمد عويس، هاجر شرف، لميس الموصلي.

ولكل من ساهم بنصيحة أو تعليق بنّاء، شكراً لكم جميعاً

الفهرس







مقدمة

" العلم مغرس كل فخر فافتخر ... واحذر يفوتك فخر ذاك المغرس

واعلم بأن العلم ليس يناله ... من همه في مطعم أو ملبس"

الإمام الشافعي

حول الكتاب – الإصدار 1.0

هذا الكتاب موجه إلى كل من يرغب بدخول مجال تطوير النظم المدمجة Embedded Systems بصورة احترافية والبدء بتعلم أساسيات هذا المجال الممتع بأسلوب عملي معتمد على التجارب.

لقد حرصت على أن يكون الشرح باللغة العربية بخطوات يسيرة ومفصلة ومع ذلك سأحافظ على استخدام بعض المصطلحات الإنجليزية في الشرح حتى تعتاد على هذه المصطلحات ويصبح من اليسير لك قراءة المراجع الإنجليزية.

الكتاب ليس مُصمم ليكون مرجع شامل بقدر ماهو موجه ليكون بداية انطلاق نحو احتراف المجال، الحقيقة أن هذا العلم لا يمكن احتواؤه أبداً في كتاب أو مرجع حتى وإن كان 1000 صفحة، لذا قمت بإضافة مصادر تعليمية بعد كل فصل تشرح المزيد من التجارب والمعلومات عن نقاط هذا الفصل فاحرص على قراءة هذه المصادر الإضافية لتستزيد من العلم.

فصول الكتاب

الفصل الأول: مقدمة سريعة عن الأنظمة المدمجة والمكونات المستخدمة في بنائها وكيفية اختيار هذه المكونات لتحقيق أقصى استفادة بأقل سعر وشرح عام لمراحل التطوير بداية من الفكرة وإنتهائاً بالمنتج الذي يباع للمستهلك.

الفصل الثاني: يقدم شرح مبسط للتركيب الداخلي للمتحكم الدقيق مع شرح لخواص ومميزات المُتحكِمات من نوع AVR وكيفية قراءة دليل البيانات Datasheet الخاصة بها واستخلاص أهم المعلومات.

الفصل الثالث: يوضح هذا الفصل الأدوات التي سنستخدمها في تطوير الأنظمة المدمجة سواء كانت العتاد "المكونات الإلكترونية" Hardware أو الأدوات البرمجية ToolChain (Softwares)

الفصل الرابع: من هنا نبدأ رحلة تعلم المُتحكِمات الدقيقة وسنبدأ مع أساسيات تشغيل أطراف المُتحكِم الدقيق وتشغيل المنافذ لتعمل كدخل أو كخرج GPIO. كما سنقوم بمجموعة من التجارب لتشغيل العناصر الإلكترونية البسيطة مثل LEDs, Switchs, 7-Segments..الخ.

الفصل الخامس: شرح لأهم القواعد والصيغ الشهير للغة السي المعيارية والمستخدمة بشكل كبير في تطوير الأنظمة المدمجة. تتميز الصيغ المعيارية بإمكانية تطبيقها على مختلف المُتحكِمات الدقيقة طالماً أن المترجم الخاص بها يدعم لغة السي.

الفصل السادس: شرح الإعدادات المتقدمة لمتحكمات AVR مثل مفهوم الفيوزات ووظائفها المختلفة مثل تغير سرعة التشغيل Clock Rate واستهلاك الطاقة، حماية البرامج الموجودة على المُتحكِم من السرقة أو التعديل وتشغيل بعض الخصائص المتقدمة الأخرى.

الفصل السابع: سنتعرف في هذا الفصل على كيفية تشغيل المقاطعات الخارجية External Interrupts وفائدة هذه الخاصية الرائعة التي تتيح صناعة تطبيقات ذات استجابة عالية السرعة للأحداث الخارجية.

الفصل الثامن: شرح أحد أشهر طرق إرسال البيانات بصورة تسلسلية بين المُتحكِمات الدقيقة والعالم الخارجي وذلك عبر بروتوكول UART والذي يعتبر أشهر بروتوكول معياري لتبادل البيانات.

الفصل التاسع: في هذا الفصل سنتعرف على كيفية قراءة الجهود الكهربية المتغيرة Analog وتحويلها إلى قيم رقمية وذلك باستخدام المحول التناظري-الرقمي المدمج داخل مُتحكِمات AVR.حيث يمكن استغلال هذا المحول في قراءة الحساسات التناظرية أو أي عنصر إلكتروني له خرج كهربي متغير.

الفصل العاشر: شرح أكواد C preprocessor حيث سنتعرف على الفارق بين الأوامر التنفيذية والأوامر التوجيهية وأهميتها بصورة مفصلة مثل الأمر #include وكذلك define وكذلك سنتعرف على كيفية صناعة المكتبات البرمجية libraries. مع شرح مثال لعمل uart driver على صورة مكتبة.

الفصل الحادي عشر: طرق استخدام أنظمة تشغيل الوقت الحقيقي Real Time OS لتشغيل المهام المتعددة Multitasking وأنظمة الاستجابة السريعة. حيث سيتم تناول نظام FreeRTOS في هذا الفصل باعتباره أفضل نظام RTOS مجاني (ومفتوح المصدر).

وفي النهاية مضاف مجموعة من المُلحقات التدريبية، كل ملحق يشرح مهارة تقنية مختلفة

الكتاب موجه خصيصاً إلى طلبة الكليات الهندسية مثل:

- ✔ تخصص هندسة الحسابات والاتصالات.
- ✔ تخصص هندسة الإلكترونيات و الكهرباء.
- ✔ تخصص ميكاترونكس.
- ✔ هواة الإلكترونيات بشكل عام.

يتطلب قراءة هذا الكتاب بعض المعرفة المُسبقة:

- ✔ أساسيات لغة السي C بشكل عام مثل استخدام المتغيرات والثوابت if - for – while
- ✔ أساسيات الإلكترونيات والكهرباء مثل المقاومات، المكثفات، البطاريات .. إلخ

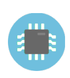
إذا لم يكن لديك اي خبرة بما سبق فأنصحك بقراءة المراجع التعليمية العربية (في نهاية الكتاب) حيث تحتوي على موارد عربية رائعة لشرح علم الإلكترونيات من الصفر

لتشغيل البرمجيات التي سنستخدمها في تطبيق الأمثلة المذكورة في الكتاب ستحتاج أن تمتلك حاسب آلي بالإمكانيات التالية على الأقل:

- معالج بنتيوم 4 أو أعلى مثل Core2Due – Core i3 – i5 - i7
- ذاكرة عشوائية RAM سعة 1 جيجا أو أكثر
- مساحة تخزينية فارغة 5 جيجا على الأقل
- نظام تشغيل Windows أو Linux (مع العلم أن التطبيق الأساسي سيكون على نظام ويندوز).

يمكنك الحصول على نسخة مجانية(مدى الحياة) من برنامج Atmel stduio من الموقع الرسمي مع العلم أن الموقع يتطلب تسجيل حساب (مجاني) لتحميل البرنامج.

http://www.atmel.com/tools/ATMELSTUDIO.aspx

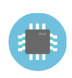


كما يمكنك الحصول على نسخة مجانية (لمدة شهر) من برنامج المحاكاة بروتس Protues من الموقع الرسمي http://www.labcenter.com

سيتم استخدام كلا البرنامجين بصورة أساسية في شرح التجارب المذكورة في الكتاب.

> ملاحظة: برامج المحاكاة مثل بروتس تتطلب استهلاك قدر كبير من الذاكرة و المعالج لذا احرص على إغلاق أي تطبيقات أخرى لا تستخدمها عند تشغيل برنامج بروتس

سيرتكز الكتاب على شرح المُتحكِمات الدقيقة من نوع **AVR – 8 bit** المُصممة بواسطة الشركة العملاقة ATmel لما لها من مميزات رائعة، وسيكون الشرح مبني على لغة السي C المعيارية أو كما تعرف باسم ANSI – C (C89, C99) وسيتم استخدام المترجم AVR-GCC المضمن مع برنامج ATmel Studioولن يتم استخدام البرامج التي تغير طرق البرمجة المعيارية مثل (Arduino IDE, Code vision, MikroC.



أتخيل أنك بعد هذه المقدمة قد تتساءل .. لماذا سيرتكز الشرح على لغة السي المعيارية ANSI – C و سبب اختياري لـ AVR بدلاً من الـ PIC ...

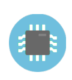
لماذا سنستخدم ANSI – C ؟

لماذا نستخدم ANSI - C بدلاً من اللغات ومعايير البرمجة الأُخرى مثل Bascom أو Flow Code **بالرغم أن هذه الطرق قد تكون أسهل في البرمجة؟** لكي نفهم الإجابة علينا أولاً أن نتعرف على كلمة ANSI وهي اختصار المعهد الوطني الأمريكي للمعايير (ANSI)

هذا المعهد قام بوضع معيار موحد للغة السي وذلك حتى تصبح الأكواد المكتوبة بها صالحة على منصات مختلفة (حتى وإن تطلبت تعديلات بسيطة). فمثلاً يمكنك كتابة برنامج بلغة السي على نظام ويندوز ومن ثم تقوم بعمل ترجمة له دون تعديل ليعمل على نظام لينكس أو العكس وباستخدام نفس المترجم Compiler.

إن تعلم لغة السي المعيارية والتدرب على تقنيات كتابة الأكواد بها **يعطيك القدرة على التعامل مع أنواع كثيرة جداً من المُتحكِمات الدقيقة**، فمثلاً بعد انتهائك من هذا الكتاب ستكون قادراً على قراءة الأكواد المكتوبة لمتحكمات ARM بدون مجهود كبير، بل قد تجد أن الأوامر شبه متطابقة في الكثير من الحالات (باختلاف أسماء المُسجِلات Registers التي سنتحدث عنها في الفصول القادمة). ليس هذا فحسب بل ستجد أن تعلم برمجة أي مُتحكِم آخر أصبحت عملية سهلة جداً طالما أن المترجم الخاص بهذا المُتحكِم يدعم السي المعيارية.

من أجل هذه الأسباب سنتعامل فقط مع البرامج التي تدعم هذه اللغة مباشرة مثل **ATmel Studio** و **CodeBlocks** أما باقي البرامج مثل **CodeVision** تجعلك تتعلم بعض الممارسات السيئة في كتابة الكود والتي قد لا تتوافق مع معايير الـ **ANSI – C** وبعض البرامج تضيف مكتبات تغير طريقة البرمجة بالكامل (مثل آردوينو) وهذا أيضاً لا أنصحك باستعماله إذا كنت تهدف احتراف تصميم الأنظمة المدمجة.



حرب المُتحكِمات - من هو الأفضل الـ AVR أم الـ PIC ؟



هذا السؤال دائماً ما يتبادر لكل من يعمل أو بدأ يدخل مجال الأنظمة المدمجة، دائماً سنجد هذا الصراع القائم بين فريق متحمس للـ AVR وآخر للـ PIC، الحقيقة أن حسم هذا الصراع أمر صعب للغاية لكن اسمح لي أن أعرفك على بعض جوانب هذه الحرب ..

> في البداية لنعترف بشيء هام، في مجال النظم المدمجة لا يوجد ما يسمى "ما هو أفضل مُتحكِم دقيق" بصورة مُطلَقة ولكن هناك "من الأنسب" للاستخدام في تطبيق معين

في بعض الأحيان نحتاج أن نصمم نظام تحكم بسعر رخيص جداً ولا نحتاج لقدرات خارقة أو مُتحكِمات متطورة لتشغيله لذا نبحث عن المُتحكِم "الأرخص" والذي يكفي فقط لهذه المهمة لذا لا تستغرب أن علمت أن المُتحكِمات (8-bit) STM8 تعتبر من أكثر المُتحكِمات مبيعاً في العالم لأنها أرخص من كل من AVR والـ PIC الـ 8 بت وتتفوق عليهم في تقديم قدرات مناسبة بسعر منخفض.

لكن دعنا نعود للسؤال الأصلي والمتسبب في حرب طويلة بين المطورين .. من الأفضل الـ AVR أم PIC ؟ للإجابة سأقوم بعقد بعض المقارنات التقنية والمالية بين مُتحكِمات كل من AVR – 8 bit و الـ PIC – 8 bit

أولاً : مقارنة السرعة

هنا سنجد أن مُتحكِمات الـ AVR – 8 bit تتفوق بفارق كبير جداً ويعتبر أداؤها أسرع بنحو 4 أضعاف من مثيلتها في الـ PIC – 8 bit وذلك لأن مُتحكِمات الـ AVR تستطيع أن تنفذ عدد أوامر في الثانية الواحدة = التردد الذي تعمل به أما الـ PIC فيمكنه تنفيذ رُبع هذا العدد

مثلاً لو معنا مُتحكِم AVR و PIC وكلاهما يعمل بتردد = 16 ميجاهرتز (16 مليون هرتز) سنجد أن الـ AVR يمكنه تنفيذ 16 مليون أمر برمجي في الثانية الواحدة Instruction per second بينما الـ PIC بنفس السرعة يستطيع أن ينفذ فقط 4 مليون أمر في الثانية الواحدة.

يرجع هذا الأمر إلى تقنية الـ Pipeline التي تتميز بها جميع مُتحكِمات الـ AVR ولا تتواجد إلا في بعض فئات الـ PIC المتطورة نسبياً.

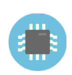
أيضاً تحتوي معظم شرائح الـ AVR على بعض الأدوات التي تسرع من تنفيذ الأوامر مثل الـ Hardware multiplier وهي وحدة معالجة لعمليات الضرب الحسابية يمكنها تنفيذ عملية الضرب في 2 نبضة فقط (سنتعرف على النبضات ومفهوم التردد في فصل الفيوزات والتحكم في سرعة التشغيل). بينما مُتحكِمات الـ PIC المماثلة لا تحتوي على هذا الأمر وقد تستغرق نفس عملية الضرب عليها نحو 40 ضعف الوقت المطلوب على الـ AVR.

ثانياً: التصميم الداخلي ومعالجة البيانات

عندما نكتب برنامج بلغة التجميع Assembly نجد فارقاً ضخم بين كليهما حيث يتمتع الـ AVR بوجود 32 مُسجِل عام Register "ريجستر" يمكن استخدامهم في معالجة وتخزين البيانات المؤقتة بسرعة وكفاءة بينما يجبرك الـ PIC على استخدام مُسجِل واحد فقط (مُسجِل التراكم Accumulator) في معظم الأوامر وهذا يعني أن البرامج المكتوبة على الـ AVR أكثر كفاءة وأسرع بكثير من البرامج المكتوبة على الـ PIC.

مثال على ذلك، البرنامج التالي مكتوب بلغة السي ومصمم لكي يبحث عن أكبر قيمة داخل مصفوفة من الأرقام Array وتم تشغيل نفس الكود على مجموعة من المُتحكِمات الدقيقة مثل ATmega16 و PIC18F و MSP

```
int max(int *array)
{
 char a;
 int maximum=-32768;
 for (a=0;a<16;a++)
          if (array[a]>maximum)
                    maximum=array[a];
return (maximum);
}
```



الجدول التالي يوضح عدد الـ cylces (نبضات الـ Clock) وعدد الأوامر بلغة الأسمبلي و سرعة التنفيذ النهائية (بالميكروثانية) للكود السابق على مختلف المُتحكِمات، لاحظ كيف أن atmega16 بالرغم من أنه يعمل بسرعة 16 ميجا إلا أنه استطاع أن يتفوق على كل من PIC16C74 وكذلك PIC18F452 الذي يعمل بسرعة 40 ميجاهرتز.

Device	Max Speed [MHz]	Code Size [Bytes]	Cycles	Execution Time [uS]
ATmega16	16	32	227	14.2
MSP430	8	34	246	30.8
T89C51RD2	20	57	4200	210.0
PIC18F452	40	92	716	17.9
PIC16C74	20	87	2492	124.6
68HC11	12	59	1238	103.2

ملاحظة: المقارنة الكاملة ستجدها داخل ملف AVR_introduction المرفق مع الكتاب وهو ملف رسمي من شركة Atmel يوضح مميزات هذه العائلة من المُتحكِمات الدقيقة

ثالثاً: استهلاك الطاقة

هنا يتفوق الـ PIC على الـ AVR بفارق واضح، حيث تتميز مُتحكِمات البيك باستهلاك منخفض للطاقة (سواء على مستوى فارق الجهد أو التيار الكهربي). ومع ذلك نجد شركة Atmel قد حسنت كثيراً بعض إصدارات الـ AVR بتقنيات استهلاك منخفضة للطاقة مثل Pico Power Save لكن ستظل مُتحكِمات البيك أفضل من الـ AVR في هذا الجانب.

رابعاً: البرمجة والدعم المجتمعي

شركة ATmel منذ بداية تصنيع الـ AVR قد اعتمدت على مترجمات compilers مفتوحة المصدر وتدعم الـ C – ANSI مباشرة مثل AVR-GCC المجاني مما تسبب في جعلها الخيار المفضل لدى الهواة والمحترفين (وهو نفس السبب الذي جعل مصممي لوحات آردوينو يختارون شرائح الـ AVR بدلاً من الـ PIC لصناعة آردوينو).

أما شركة Microchip فقد اتخذت مساراً مُختلِف، حيث نجد أن برنامج MPLAB يخالف الـ ANSI – C خاصة عند كتابة برامج لعائلات مثل PIC16F مما يجعل تعديل الأكواد المكتوبة بها لاستخدامها مرة أخرى أو نقلها لمتحكمات أخرى عملية صعبة.

هنا مجدداً يتفوق الـ AVR، كما أنه هناك دليل واضح أيضاً على التفوق القوي وهو مدى كبر حجم "مجتمع" الهواة والمطورين والمواقع الإلكترونية الأجنبية التي تدعم الـ AVR والتي لن تجد مثلها في حالة ال PIC.

خامساً: السعر مقارنة بالمميزات المدمجة

في الأسواق المحلية تعتبر مُتحكِمات ال AVR وال PIC متقاربة جداً في السعر لنفس العائلات (**عائلة المُتحكِمات**: هي مجموعة من المُتحكِمات الدقيقة التي تشترك في خصائص وإمكانيات مشتركة مثل سعة الذاكرة أو الحجم أو الطاقة المستهلكة .. الخ) فمثلاً نجد في السوق المصري أن سعر ال **ATmega16** مساوي تقريباً لـ **PIC16F877a** (25 جنية مصري وقت كتابة هذه السطور وهو ما يساوي 3.5 دولار)

لكن نجد أن ATmega16 يوفر قدرات مُضاعفة مقارنة بسعر Pic16F منها مثلاً: الأتميجا أسرع 4 مرات من البيك + توفير نحو 3 أضعاف عدد مخارج الـ PWM ونحو ضعف معدل سحب التيار لكل طرف من أطراف المُتحكِم كما أن الذاكرة في الـ ATmega16 تساوي مرة ونصف حجم الذاكرة في ال PIC16F877.

يجب التنوية أن هذه الأسعار هي **أسعار محلية** وقد تختلف من دولة لأخرى أو عند الشراء بكميات كبيرة

نستنتج من ذلك أنه في حالة الرغبة بتطوير نظام سريع الاستجابة أو يقوم بعمليات حسابية معقدة وبسعر مناسب فإن ال AVR هو الخيار الأمثل لأغلب الأنظمة المدمجة المعتمدة على المُتحكِمات 8 بت الرخيصة

سادساً: التوافر الكمي في الأسواق

هنا نجد أن شركة MicroChip (المصنعة لـ PIC) تتفوق على ATmel فكلا السوقين المحلي والعالمي نجد أن منتجات Microchip متوفرة ويسهل الوصول إليها مقارنة بال AVR .

هذه هي أهم الأسباب التي قد تجعلك تفضل الـ AVR عن البيك وقد تحسم الصراع بين المُتحكِمات ال 8 بت، لكن مجدداً تذكر أنه في بعض الحالات يكون عليك اختيار مُتحكِم لأنه الأنسب والأفضل سعراً.

وتستمر الحرب مع مقارنات إضافية

إذا أحببت أن تقرأ المزيد عن حرب المقارنات بين الـ AVR والـ PIC فعليك بهذه المقارنات الرائعة والتي ستوضح جوانب إضافية من هذه المقارنات

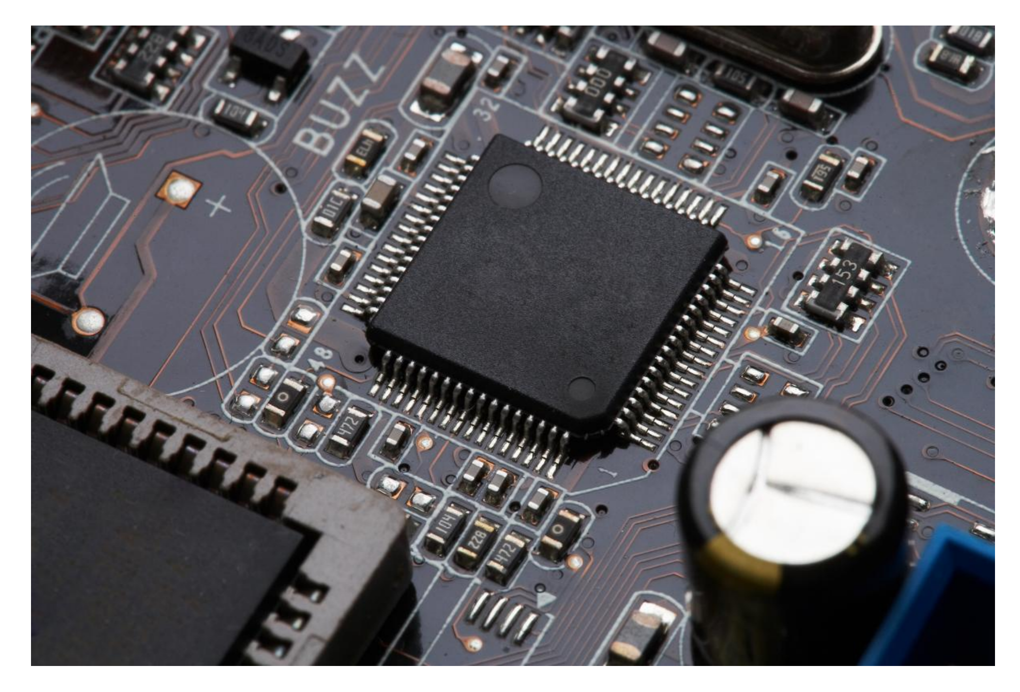
- www.kanda.com/pic-vs-avr.php
- www.youtube.com/watch?v=DBftApUQ8QI
- arstechnica.com/civis/viewtopic.php?f=11&t=409115
- stackoverflow.com/questions/140049/avr-or-pic-to-start-programming-microcontroller

الفصل الأول

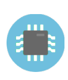
" هؤلاء الذين يمتلكون الجنون الكافي ليؤمنوا أن بإمكانهم تغيير العالم، هم من يغيرونه بالفعل "

ستيف جوبز – شريك مؤسس لشركة Apple

1. مُقدمة عن الأنظمة المُدمجة



- ✔ معنى النظام المدمج
- ✔ مكونات الأنظمة المدمجة
- ✔ مراحل تطوير المنتجات الإلكترونية المعتمدة على الأنظمة المدمجة
- ✔ كيفية اختيار المُتحكِم الدقيق المناسب



1.1 معنى النظام المدمج Embedded System

النظام المدمج أو كما يسمى في بعض الأحيان "النظام المُضمن" هو أي نظام حاسوبي صغير الحجم يقوم بمجموعة من الوظائف التي تخدم أداة أو منتج معين، وغالباً لا تباع هذه الأنظمة المدمجة للناس مباشرة ولكنها تكون "مدمجة Embedded" مع منتج معين، فمثلاً عند شراء سيارة حديثة أو فرن ميكروويف أو غسالة كهربائية أو حتى مكيف هواء فإنك ستجد أن جميع هذه المنتجات أصبحت تحتوي على حواسيب صغيرة تقدم وظائف تحكم ذكية مما يجعل كل المنتجات السابقة تحتوي على نظم مدمجة.

تستخدم الأنظمة المدمجة في مجموعة واسعة جداً من التطبيقات، أشهرها:

- **التحكم الآلي** مثل الأنظمة المدمجة الموجود في المصانع، الطائرات، الصواريخ والأقمار الصناعية وأي ماكينة تعمل بصورة تلقائية (أوتوماتيكية) هذه الأنظمة جميعها تصُمم لغرض واحد فقط وهو التحكم في منتج معين.
- **المنتجات الخدمية** مثل المنتجات التي عادة نشتريها لأنفسنا في المنزل أو المكتب مثل مكيف الهواء أو الميكروويف الذي يحتوي على نظام تحكم إلكتروني في الحرارة.
- **المنتجات الترفيهية** مثل منصات الألعاب Xbox, Gameboy, Wii وكذلك المنتجات التي أصبحت تحمل وصف "ذكية" مثل الهواتف الذكية، الساعات الذكية وحتى أنظمة التلفاز الحديثة جميعها تعتبر أنظمة مدمجة.

- **أنظمة الاتصالات الحديثة** والتي لها نصيب كبير من هذا المجال خاصة بعد ظهور تقنيات الاتصال اللاسلكي مثل Bluetooth والـ Wifi حيث تحولت الأجيال القديمة من أنظمة الاتصالات التي كانت تعتمد على الـ Analog Electronics إلى تقنيات المعالجة الرقمية المعتمدة على الأنظمة المدمجة فمثلاً جميع أجهزة الموجهات Routers التي توفر لنا الإنترنت ما هي إلا Embedded Linux Systems وكذلك أنظمة الراديو القابلة للبرمجة SDR وشبكات المحمول هي أيضاً نوع من الأنظمة المدمجة عالية الأداء.

1.2 مكونات النظام المدمج

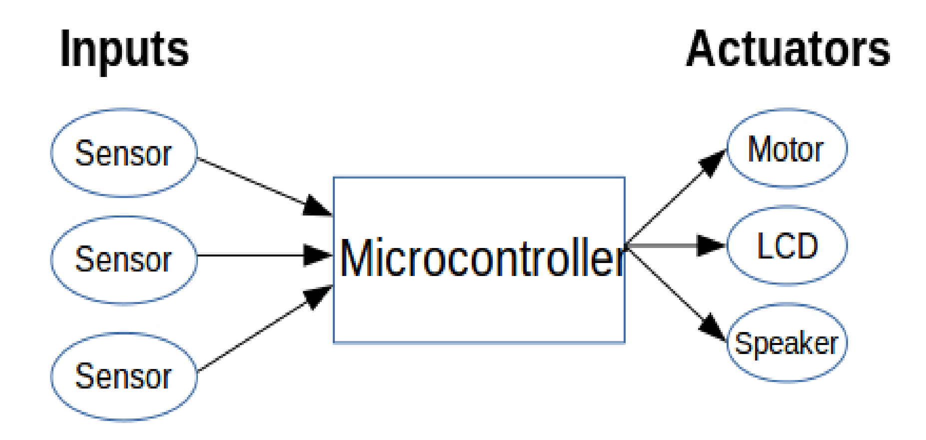
عادة تتكون النظم المدمجة من 3 مكونات رئيسية

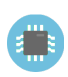
- **المُتحكِم الدقيق MicroController** والذي يعتبر العقل المُتحكِم في النظام.

- **أدوات الإدخال Input devices** مثل الحساسات المختلفة، أزار الضغط أو أي وسيلة إدخال معلومات للمُتحكِم.

- **أدوات إخراج Output devices** والتي تُسمى في بعض الحالات Actuators وتعتبر كل ما يتحكم به الـ Microcontroller مثل المُحركات Motors، الشاشات LCD، سماعات صوتية ... الخ.

يتم اختصار أدوات الإدخال والإخراج بكلمة I/O وهي اختصار (Input/Output Devices)

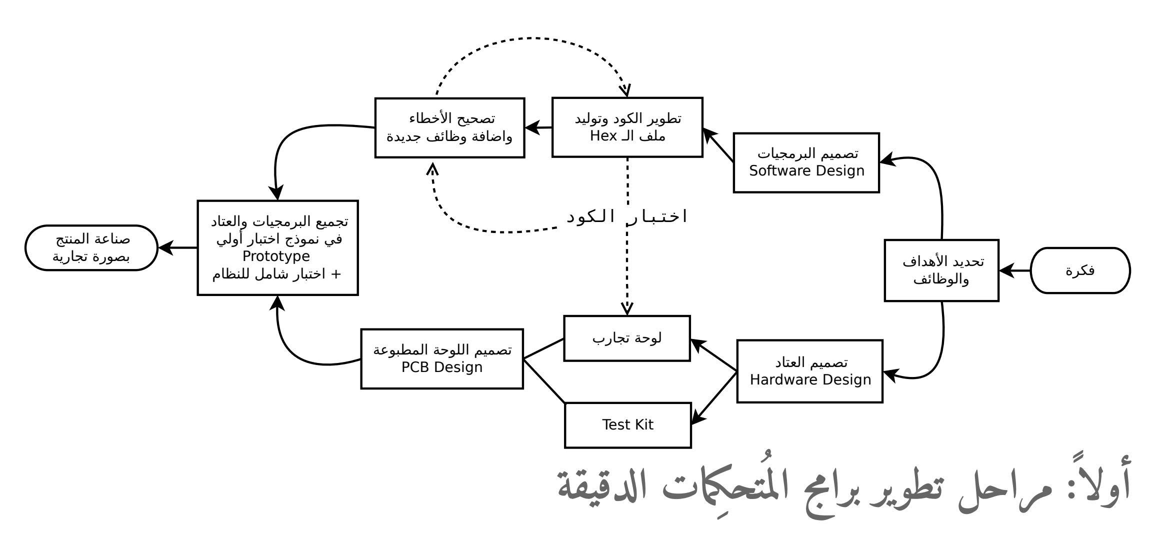




1.3 مراحل تطوير الأنظمة المدمجة

قبل أن نبدأ تعلم صناعة الأنظمة المدمجة علينا أن نفهم الخطوات التي تساعدك على التخطيط لمشروع ناجح وفعال، فمثلاً إذا جاءتك فكرة لجهاز رائع، كيف تنفذها؟ ما هي الأدوات التي ستستخدمها؟ ما هي مراحل تطوير المشروع لتصل إلى منتج نهائي؟..

الصورة التالية توضح الخطوات التي يتبعها مصممو الأنظمة المدمجة في تطوير أي منتج بداية من الفكرة حتى صناعة المنتج بصورة تجارية، كما نرى هناك مساران أساسيان وهما تصميم الـ software وتصميم الـ Hardware.



أولاً: مراحل تطوير برامج المُتحكِمات الدقيقة

مثل جميع أنظمة الحواسيب في العالم نجد أن المُتحكِمات الدقيقة لا يمكنها أن تعمل دون برنامج يكتب بداخلها وهذا البرنامج يجب أن يكتب بالصيغة الثنائية الرقمية Binary فقط الصفر والواحد، هذه الصيغة غير مناسبة للفهم بالنسبة للبشر ويصعب تفسيرها. لذا تقوم الشركات المصنعة للمعالجات والمُتحكِمات الدقيقة بصناعة بعض الأدوات البرمجية التي تُسهل على المطورين أن يصنعوا برامج بلغات مفهومة وقابلة للقراءة.

في البداية كانت الشركات تصمم برمجيات التجميع Assemblers التي توفر للمطور مجموعة من الأوامر تسمى بأوامر التجميع Assembly Instructions .

والتي كانت أوامر قصيرة وسهلة نسبياً مثل ADD (اجمع رقمين) أو SUB (اطرح رقمين) ,ولكن كان هناك عيوب كثيرة لكتابة البرامج بهذه اللغة مثل الحجم والوقت، حتى أن بعض البرامج كانت تصل إلى عشرات الآلاف من السطور. وكان هناك مثل شهير يقول **"كتابة برنامج معقد بلغة الأسمبلي موازي لحفر أساسات ناطحة سحاب باستخدام ملعقة"**.

ظل الأمر هكذا فترة من الزمن حتى ظهرت اللغات عالية المستوى **High level language** مثل لغة السي. وهي لغات تُسهل كتابة الكود البرمجي وتحويله إلى لغة الآلة تلقائياً عن طريق المترجمات Compilers وبذلك أصبحت عملية تطوير الكود أسهل بكثير.

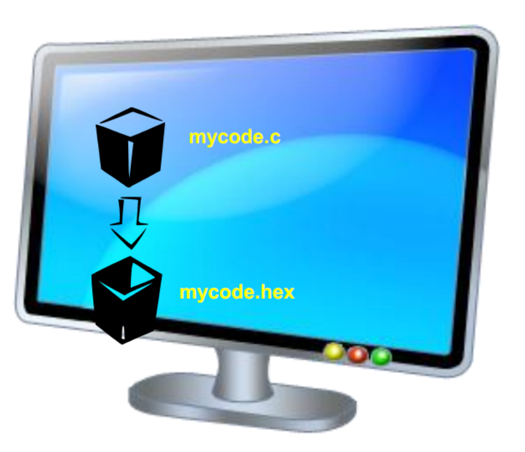
باستخدام لغة السي يمكننا تطوير برامج المُتحكِمات الدقيقة كالتالي:

1. **كتابة البرنامج بلغة السي:** في هذه المرحلة نستخدم لغة السي للتعبير عن الوظائف التي نريد تنفيذها من المُتحكِم الدقيق
2. **توليد ملف الـ Hex :** ملف الهيكس هو الملف الذي يحتوي على البرنامج الحقيقي الذي سيخزن داخل ذاكرة المُتحكِم ويتم توليده تلقائياً من تحويل الكود المكتوب بلغة السي إلى الأوامر البرمجية بصيغة الـ hex عن طريق الـ toolchain (سنتحدث عنها بالتفصيل في الفصل التالي).

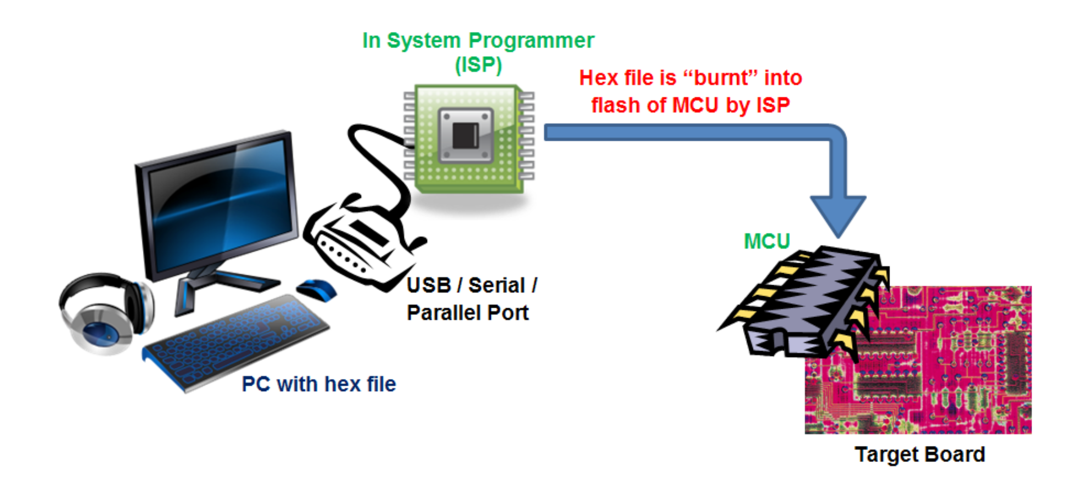
```
#define F_CPU  16000000UL
#include "servo.h"
int main( void )
{
        cli();
        Init_servo_out();
        sei();

        for (;;)
        {
        //servo_channel_A_updated = 1;
```

```
:100000000C942A000C9447000C9447000C94470071
:100010000C9447000C9447000C9447000C94470044
:100020000C9447000C9447000C9447000C94470034
:100030000C9447000C9447000C9447000C94470024
:100040000C9447000C9447000C9447000C94470014
:100050000C94470011241FBECFE5D8E0DEBFCDBF12
:1000600010E0A0E6B0E0E8EDF0E002C005900D92EF
:10007000A036B107D9F710E0A0E6B0E001C01D92AC
:10008000A036B107E1F70E9453000C946A000C946B
```



3. **رفع البرنامج من الحاسوب إلى ذاكرة المُتحكِم:** هذه المرحلة التي يتم كتابة (أو كما يسميها البعض بعملية حرق burn) البيانات الرقمية داخل ذاكرة المُتحكِم ليبدأ بتنفيذها حيث يقوم برنامج الرفع uploader بقراءة ملف الهيكس وتحويل القيم المسجلة بداخله إلى بيانات ثنائية binary ثم يقوم بكتابتها داخل العناوين المخصصة لها في ذاكرة المُتحكِم.



4. **اختبار البرنامج واكتشاف الأخطاء:** في هذه المرحلة يتم تشغيل المُتحكِم الدقيق على لوحة التجارب أو على Test Kit للتأكد من أن البرنامج ينفذ المطلوب أو لاكتشاف أي أخطاء، وقد يتم تكرار هذا الأمر عشرات المرات حتى نصل إلى برنامج يؤدي جميع الوظائف المطلوبة منه بأقل نسبة خطأ.

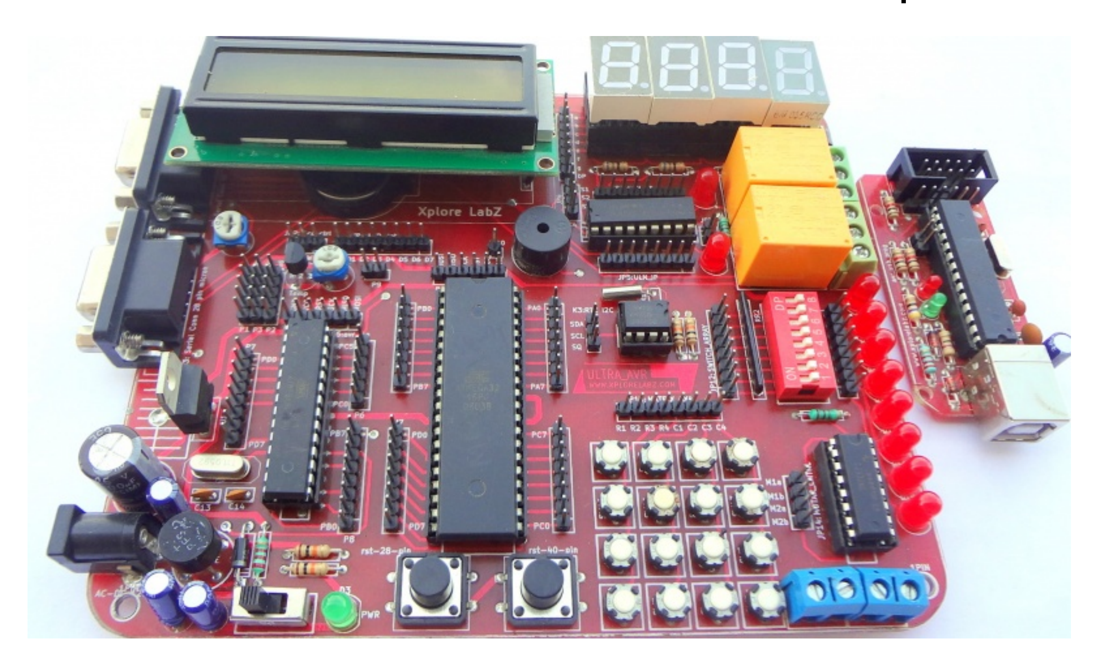
للقيام بكل ما سبق سنحتاج لمجموعة من الأدوات البرمجية والمكونات الإلكترونية وهو ما سيتم شرحه بالتفصيل في الفصل التالي.

ثانياً: مراحل تطوير العتاد

لتطوير أي مشروع سنحتاج أن نوصل المُتحكِم الدقيق بالمكونات الإلكترونية التي سيتحكم بها وهو ما يعرف بمفهوم الـ Devices Interfacing (مواجهة الأجهزة المختلفة) فالمتحكم الدقيق لا يعمل بمفرده وإنما يحتاج أجهزة أخرى ليستقبل منها قراءات (مثل الحساسات Sensors) أو ليتحكم بها مثل الشاشات والمحركات.

هناك طريقتين أساسيتين لعمل ذلك وهما، استخدام اللوحات التطويرية Development Kit أو استخدام لوحة التجارب Breadboard، كل طريقة لها مميزات وعيوب.

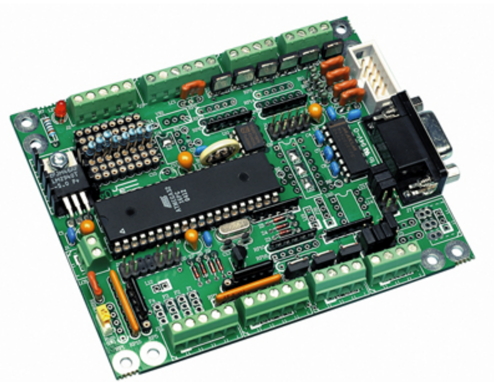
الـ Development Kit

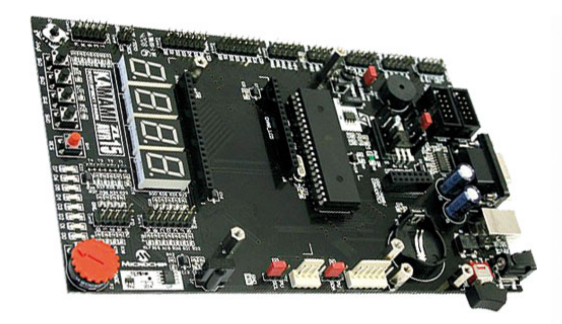


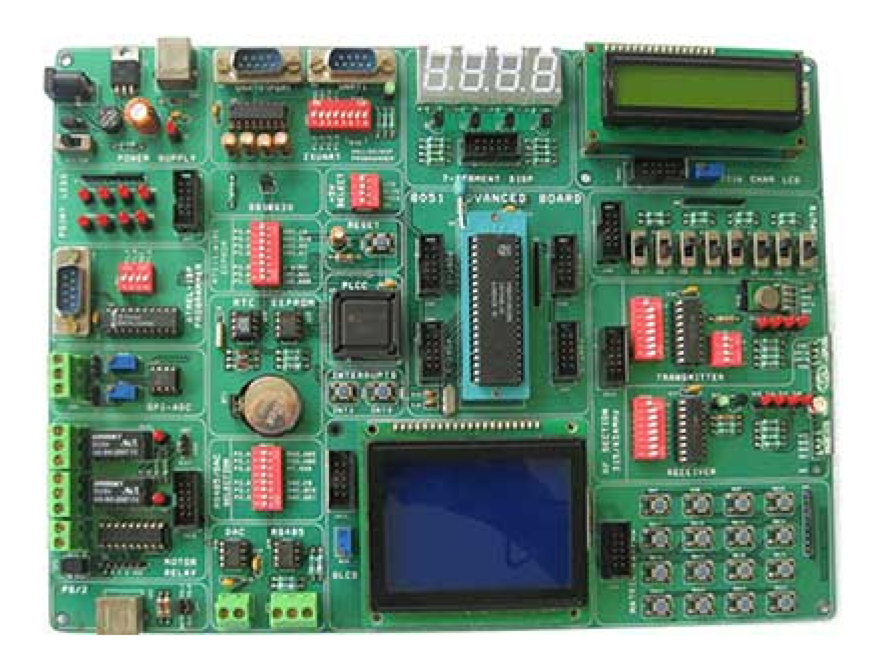
هي لوحة اختبار مكونة من المُتحكِم الدقيق + مجموعة كبيرة من العناصر الإلكترونية المتصلة به بصورة جاهزة للتشغيل مثل شاشة LCD، لوحة مفاتيح، أزرار تحكم، حساسات حرارية وضوئية، ريلاي Relay وبعض أدوات الاتصال الرقمية مثل مُحول RS232 وقد يوجد بها أكثر أو

أقل من ذلك. هذه اللوحات تُسهل عملية التطوير بصورة كبيرة فهي تحتوي على معظم ما قد تحتاجه على لوحة واحدة جاهزة ومتصلة ببعضها البعض وبالتالي لن تحتاج لشراء مكونات أخرى أو توصيل عناصر إضافية وستوفر عليك وقت بناء الدوائر الإلكترونية.

الصور التالية هي لمجموعة مختلفة من الـ Development kits







مرفق مع الكتاب مجلد يحتوي على تصميمات لمجموعة لوحات تطويرية مفتوحة المصدر مخصصة لـ AVR أغلبها مصمم للمُتحكِم الدقيق ATmega32/ATmega16 و يمكنك صناعتها بنفسك بتكلفة أقل من شرائها.

لوحة التجارب Breadboard

الطريقة الثانية هي استخدام لوحة التجارب البلاستيكية والتي تساعدك على بناء أي دائرة إلكترونية باستخدام الأسلاك، تتميز هذه اللوحة بأنه يمكنك بناء أي دائرة قد تخطر على بالك فمن السهل أن تفك وتركب أي عنصر أو شريحة إلكترونية (من نوع DIP) على هذه اللوحة، سيتم استخدام هذه الطريقة في الكتاب لأنها الخيار الأرخص والأكثر توافراً في كل البلاد العربية.

معلومة إضافية: الشرائح الإلكترونية الـ DIP (Dual in-line Package) هي التي تمتلك صفين من الأرجل المعدنية والتي يمكن توصيلها بثقوب على لوحة التجارب أو الـ PCB أما SMD وهي اختصار لكلمة Surface Mount Device هي الشرائح صغيرة الحجم و تمتلك أرجل معدنية صغيرة جداً ويتم لحامها على سطح الـ PCB فقط

ATmega32 (SMD

ATmega32 (DIP)

PDIL40 PDIL28 PDIL20 SOIC20 PDIL8 SSOP20 SOIC8

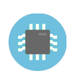
PLCC44 TQFP64 TQFP44 MLF64 MLF32

صور مختلفة لأنواع تغليف مُتحكِمات AVR بجميع أنواع وأحجام الـ DIP والـ SMD

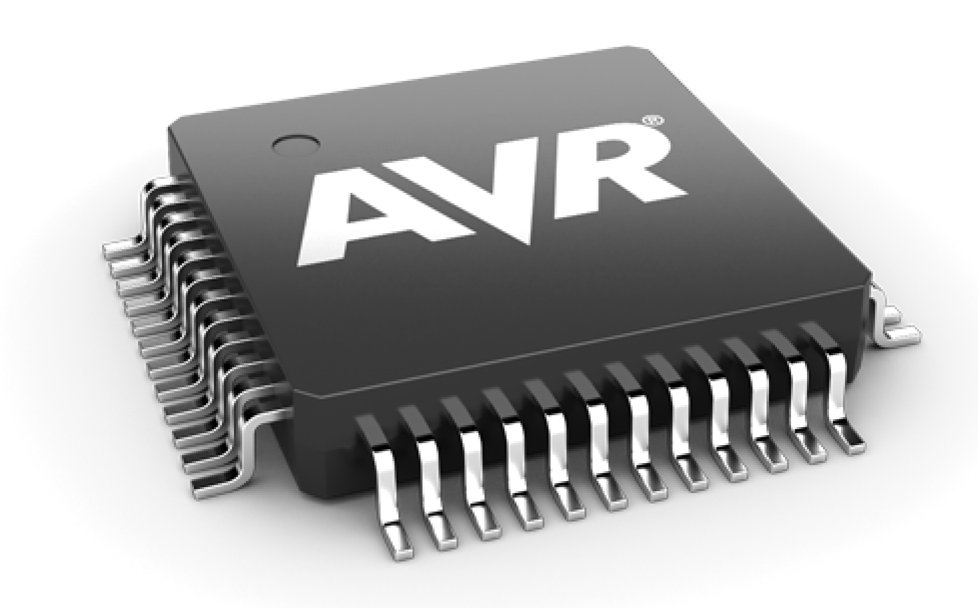
الفصل الثاني

"نحن نِتاجِ ما نفعله بتكرار، الجودة إذاً ليست عَملاً
بل هي مُنتج العادات الجيدة "

أرسطو – فيلسوف يوناني



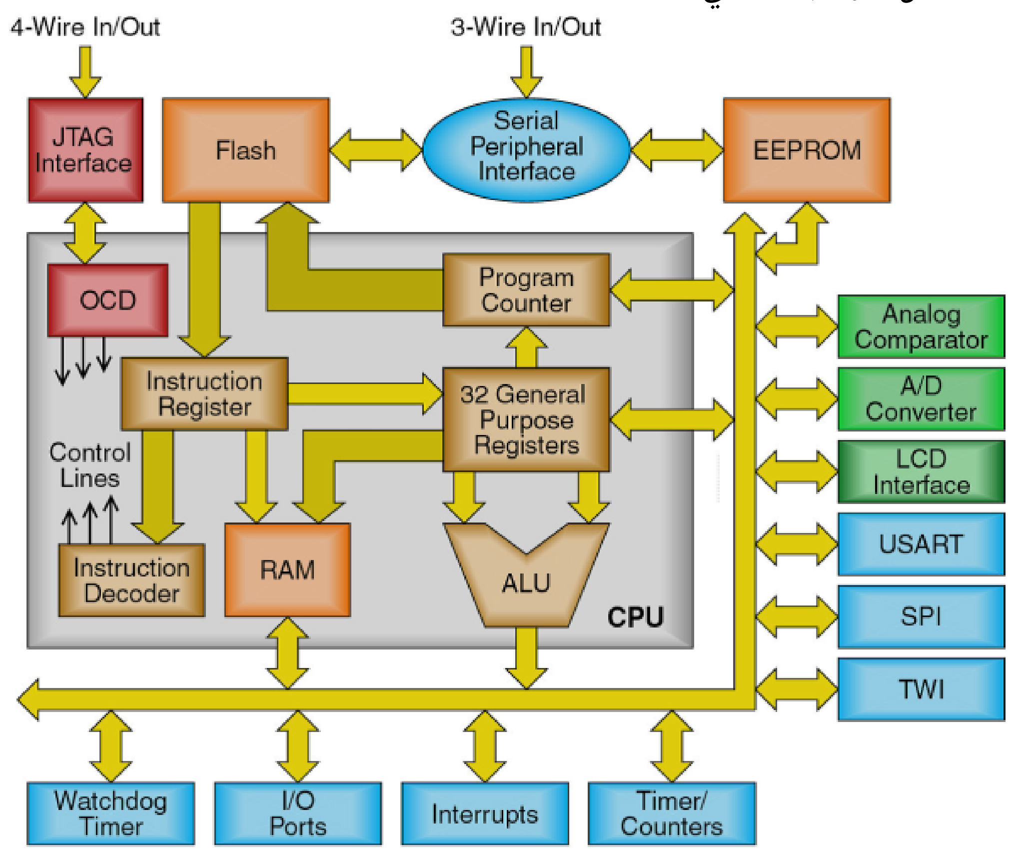
2. نظرة عامة على مُتحكِمات AVR



- ✔ تركيب المُتحكِم الدقيق
- ✔ مميزات معمارية AVR
- ✔ كيف تختار المُتحكِم المناسب من عائلات AVR المختلفة
- ✔ مقدمة عن قراءة دليل البيانات Datasheet
- ✔ نظرة عامة على المُتحكِم ATmega16
- ✔ نظرة عامة على المُتحكِم ATTiny84

2.1 تركيب المُتحكِم الدقيق ومعمارية AVR

المُتحكِم الدقيق هو حاسوب متكامل على شريحة واحدة Computer On Chip يُستخدَم في التحكم بمجموعة من الأجهزة الأخرى. ومثل جميع الحواسيب يحتوي المُتحكِم الدقيق على نفس مكونات الداخلية للحاسب الآلي ولكن بقدرات مختلفة من حيث الكم والقوة. الصورة التالية تمثل التركيب الداخلي لـ AVR



المُعالج Micro-processor قلب المُتحكِم الدقيق ويتكون من وحدة الحساب والمنطق ALU المسؤولة عن جميع العمليات الحسابية والمنطقية مع وحدات قراءة الأوامر من الذاكرة+ مجموعة من المُسجِلات العامة والخاصة Register والتي سنتعرف عليها في الفصول القادمة. يتم التحكم في سرعة المعالج من خلال دائرة المذبذب Oscillator حيث تساوى سرعة المعالج عدد النبضات الناتجة من دائرة المذبذب (التردد).

الذاكرة ثابتة وتنقسم إلى نوعين، الأول: يُستخدَم في تخزين البرامج وتسمى بالفلاش Flash Memory وهي تماثل الهارديسك Hard-disk في الحاسب الشخصي ويحب البعض أن يسميها Program memory. أما النوع الثاني فتسمى الـ EEPROM وهي ذاكرة مخصصة لتخزين القيم الصغيرة والهامة مثل بعض الثوابت أو المتغيرات التي تؤثر في برنامج المُتحكِم ويجب أن تحفظ من الضياع.

الذاكرة مؤقتة SRAM وهي اختصار لـ Static RAM، وهي تماثل الذاكرة العشوائية الموجودة في الحواسيب الشخصية التي نستخدمها.

المُسجِلات Registers وهي أحد صور الذاكرة وتماثل الذاكرة المؤقتة من حيث التركيب وطريقة العمل ولكنها تستخدم في التحكم بجميع إعدادات ووظائف المُتحكِم الدقيق كما سنرى في الفصول القادمة تبعاً.

الوحدات الطرفية (peripheral units (sets وهي مجموعة من الوحدات التي تساعد المُتحكِم الدقيق في أداء وظيفته الأساسية (وهي التحكم بأجهزة أخرى) من أمثال هذه الوحدات، المنافذ العامة PORTS، المحول التناظري الرقمي ADC، المؤقتات Timers، وحدات الاتصال ومعالجة البيانات التسلسلية مثل USART, SPI, i2C و بعض مُتحكِمات AVR قد تحتوي على أنظمة للتشفير مدمجة بداخلها **CryptoAuthentication** .. الخ.

ما الفرق بين الذاكرة العشوائية (المؤقتة) داخل الحواسيب الشخصية والمُتحكِمات الدقيقة ولماذا تسمى Static وليس Dynamic؟

الفارق الأساسي هو العنصر الذي تصنع منه الذاكرة، حيث تتميز ذاكرة المُتحكِمات بأنها تصنع من الـ Flip-Flop الذي يتميز بالقدرة على الاحتفاظ بالبيانات بأقل تيار كهربي ممكن وبالتالي فهو الخيار الأفضل من ناحية الاستهلاك للطاقة كما أن البيانات الموجودة عليه لا تحتاج لعملية تجديد Refreshing مثل الذاكرة "dynamic RAM" الموجودة في الحواسيب التقليدية والتي تصنع من المكثفات الطفيلية Parasitic Capacitors والتي تحتاج دائماً لعملية تجديد Refreshing وإلا تضيع البيانات المخزنة بداخلها مع مرور الوقت (أكثر من 10 مللي ثانية كفيلة باختفاء البيانات من DRAM) كما أنها تستهلك الكثير من الطاقة بسبب هذه العملية.

ما الفارق بين الـ Flash والـ EEPROM بالرغم أن كلاهما تقنياً يعتبر EEPROM؟

الـ EEPROM هي اختصار لعبارة Electrical Erasable Programmable Read Only Memroy والتي تعني الذاكرة التي تستخدم للقراءة فقط ويتم برمجتها كهربياً.

المُتحكِمات الدقيقة غالباً ما تحتوي على نوعين من الـ EEPROM الأولى تسمى الـ Flash لأنها سريعة جداً في كتابة البيانات وقد تصل سرعة الكتابة عليها إلى واحد MegaBit/S فمثلاً قد تكتب 1 بايت بداخل الفلاش في زمن 1 ميكروثانية فقط. بينما الـ EEPROM التقليدية بطيئة للغاية مقارنة بالفلاش حيث أن كتابة 1 بايت بداخلها قد يستغرق 1 مللي ثانية (يعني أبطأ بنحو 1000 مرة من الـ Flash).

2.2 مميزات معمارية الـ AVR

يقصد بكلمة معمارية Architecture طريقة توصيل المكونات الداخلية للمُتحكِم مع بعضها البعض ومدى حجم البيانات التي تستطيع هذه المكونات أن تعالجها. فمثلاً جميع مُتحكِمات AVR يوجد بها المكونات السابق ذكرها وبينها العديد من الأشياء المشتركة. لكن سنجد أنه هناك اختلافات رئيسية بين العديد من مُتحكِمات الـ AVR

بعض الخصائص المشتركة بين جميع مُتحكِمات AVR

- **معمارية Harvard** هذه المعمارية الحديثة نسبياً تعني أن المعالج المركزي يستطيع أن يتواصل مع الذاكرة RAM و الـ ROM في نفس الوقت حيث نجد أن جميع مُتحكِمات الـ AVR تستطيع أن تكتب في الـ RAM وتقرأ من الـ ROM في نفس اللحظة. على عكس المعماريات القديمة مثل Von Neumann والتي تسمح للمعالج أن يقوم بعمل شيء واحد فقط (إما القراءة أو الكتابة في نفس اللحظة).
- **Single Cycle Execution** معظم مُتحكِمات AVR تمتلك القدرة على تنفيذ أوامر برمجية = سرعة المعالج فمثلاً إذا كان تردد المعالج = 16 ميجا (16 مليون نبضة) فهذا يعني أن المُتحكِم يستطيع أن ينفذ 16 مليون أمر في الثانية الواحدة. ويرجع

الفضل إلى وجود نسختين من أنظمة قراءة الذاكرة وفك تشفير الأوامر 2x program counter + 2x instruction decoder وكلا النسختين تعملان معاً في نفس الوقت مما يضاعف سرعة وعدد الأوامر التي يتم نسخها من الذاكرة.

- **Self programming memory** وتعتبر أحد مميزات الـ AVR الرائعة والتي تعني إمكانية استخدام الذاكرة الفلاش لتخزين المتغيرات أثناء تشغيل المُتحكِم (كأنها تقوم بوظيفة الـ EEPROM التقليدية) حيث يمكن استخدام بعض الأوامر البرمجية لتغير محتوى ال Flash memory أثناء تشغيل المُتحكِم وبدون استخدام أي مبرمجة خارجية (burner) Programmer. يمكنك معرفة كافة التفاصيل من الملف الذي أصدرته شركة ATmel ويشرح جميع الأوامر البرمجية لهذه الميزة الرائعة (بلغة السي). http://www.ATmel.com/Images/doc2575.pdf

تختلف المُتحكِمات فيما بينها على حسب الـ peripheral units الموجودة بداخلها وتقنية معالجة البيانات سواء كانت 8 أو 16 أو 32 بت.

ما معني 8 بت أو 32 بت؟

يعبر هذا الرقم عن حجم البيانات الذي يستطيع المعالج المركز CPU داخل المُتحكِم الدقيق أن يتعامل معه في النبضة الواحدة.
فمثلا إذا كان المُتحكِم من نوع 8 بت فإنه يستطيع أن يجمع رقمين 8 بت مع بعضهم في نبضة الواحدة. لكن إذا جعلت المعالج يجمع رقمين بطول 16 بت فإنه سيضطر أن يتعامل مع الأرقام على أكثر من مرة بحيث يجزأ الأرقام إلى مجموعات 8 بت فقط. أما المُتحكِمات ال 32 بت تعني أن المعالج يمتلك القدرة على القيام بجميع العمليات الحسابية والمنطقية على بيانات بطول 32 بت في النبضة الواحدة.

2.3 كيف تختار بين عائلات الـ AVR المختلفة

تعتبر مهارة اختيار المُتحكِم المناسب من أهم ما يجب أن يتعلمه أي مهندس نظم مدمجة. حيث أن الشركات المنتجة للمتحكمات الدقيقة عادة ما تصنع المئات من المُتحكِمات الدقيقة وتقسمها إلى عائلات تختلف فيما بينها على حسب السعر والإمكانيات لكل مُتحكِم. لذا سيتوجب عليك أن تتقن إختيار المُتحكِم المناسب لأداء أفضل تصميم بأقل سعر ممكن.

تعتبر أهم العوامل المؤثرة في تصنيف المُتحكِمات الدقيقة هي:

سرعة معالجة البيانات والإستجابة المطلوبة	عدد أطراف التحكم العامة GPIO والتي تمثل عدد المداخل والمخارج	عدد وإمكانيات الـ Peripheral Devices المتوفرة للمُتحكِم	مساحة الذاكرة المطلوبة والتي ستحدد حجم البرنامج الذي سيوضع على المُتحكِم	المجال أو البيئة التي سيعمل بها المُتحكِم مثل درجة الحرارة و معدل استهلاك الطاقة

على حسب العوامل الخمسة السابقة سنجد أن شركة ATmel قسمت مُتحكِمات AVR إلى 6 عائلات أساسية منها أربعة عائلات عامة General purpose microcontrollers مما يعني أنه يمكن استخدامها لجميع مجالات النظم المدمجة ومختلف المنتجات. وهناك عائلتين مصممتين لمنتجات محددة فقط:

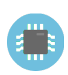
العائلات العامة General purpose microcontrollers

megaAVR – 8 bit

ATTiny – 8 bit

AVR Xmega – 8 & 16 bit

AVR - 32 bit



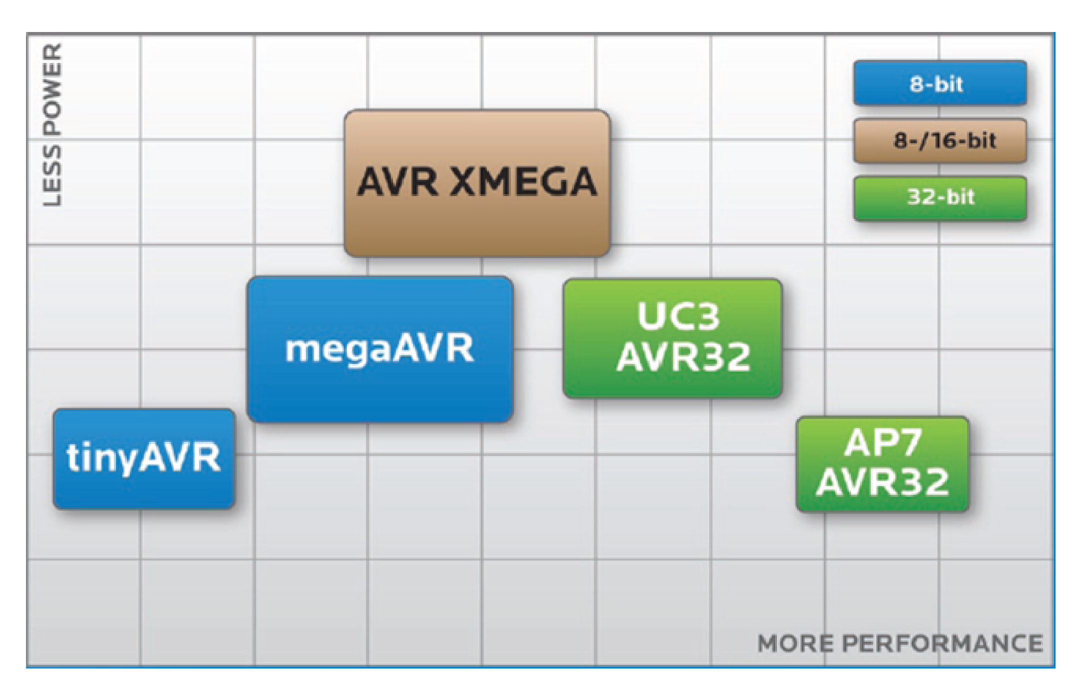
العائلات المُتخصصة Special purpose microcontrollers

Automotive AVR هذه العائلة مُصممة للأنظمة المدمجة الخاصة بالسيارات لذا فهي تتميز بتحمل الظروف القاسية مثل درجات الحرارة المرتفعة (يمكنها العمل في بيئة تصل درجة حرارتها إلى 150 درجة مئوية)، كما تحتوي على نظم تشفير خاصة لحماية المعلومات المخزنة بداخلها وكذلك تحتوي على أنظمة حماية من مشاكل التيار الكهربي وفرق الجهد (مثل حدوث قصر في الدائرة short circuit)

Battery Management عائلة المُتحكِمات الخاصة بإدارة وتشغيل البطاريات، ومصممة لتدير عملية الشحن والتفريغ الآمن للبطاريات

> سيرتكز الكتاب على شرح العائلة الأولى والثانية mega, ATTiny باعتبارهم أشهر العائلات وأكثرها توافراً على مستوى العالم

الصورة التالية توضح ترتيب القوة والمميزات التي تحتويها كل عائلة، حيث نجد أن الخط الأفقي يعبر عن قوة الأداة Performance والخطأ الرئسي يعبر عن مدى انخفاض استهلاك الطاقة.



عند الضغط على أي من أسماء العائلات بالأعلى ستنتقل إلى صفحة على موقع Atmel توضح جميع أفراد هذه العائلة من المُتحكمات مع عرض سريع لخصائص كل مُتحكِم مثل حجم الذاكرة وعدد أطراف المُتحكِم. هذه الصفحة تقدم مقارنة سريعة بين المُتحكِمات لتساعدك على اختيار المُتحكِم الأنسب لمشروعك.

عند الضغط على اسم أي من المُتحكِمات مثل ATmega16 سيتم نقلك إلى صفحة المُتحكِم والتي تحتوي على جميع البيانات المتعلقة بهذا المُتحكِم بما في ذلك أهم ملف وهو **"دليل البيانات Datasheet"** والذي يتوفر منه نسختين، summery "مختصر سريع" أو الدليل الكامل complete.

Device	Description
ATmega168PB	8-bit AVR Microcontroller, 16KB Flash, 32-pin
ATmega48	8-bit AVR Microcontroller, 4KB Flash, 28/32-pin
ATmega48A	8-bit AVR Microcontroller, 4KB Flash, 28/32-pin
ATmega48P	8-bit picoPower AVR Microcontroller, 4KB Flash, 28/32-pin
ATmega48PA	8-bit picoPower AVR Microcontroller, 4KB Flash, 28/32-pin
ATmega48PB	8-bit Atmel® AVR® Microcontroller, 4KB Flash, 32-pin
ATmega88PB	8-bit Atmel® AVR® Microcontroller, 8KB Flash, 32-pin
ATmega8	8-bit AVR Microcontroller, 8KB Flash, 28/32-pin

في هذا الكتاب دائما سنستخدم الدليل الكامل من أي Datasheet. لذا قم بتحميل كل من الدليل الخاص بالمتحكم ATmega16 و ATtiny84

من خلال الصفحات السابقة وملفات الـ Datasheet يمكنك تحديد المُتحكِم الدقيق الذي يمتلك الإمكانيات المناسبة للمشروع الذي تريده. بالتأكيد اختيار المُتحكِم يجب أن يكون مقترن بخبرتك في مجال البرمجة وتحسين الأكواد المكتوبة للاستفادة القصوى من المُتحكِم. لذا سنجد أن مهارة اختيار المُتحكِم المناسب ستزداد عندما تتقن برمجة هذا النوع من المُتحكِمات.

2.4 قراءة دليل البيانات Datasheet

تساعدك الـ Datasheet على فهم المُتحكِم الدقيق بصورة مفصلة فهي تحتوي على طريقة تشغيله وبرمجته، وتحتوي أيضاً على جميع البيانات التقنية المتعلقة بالمتحكم مثل: التصميم الداخلي، وظائف الأطراف، المُسجِلات، الطاقة، تقنيات البرمجة، كيفية تفعيل القدرات التي يملكها المُتحكِم أو إلغائها … الخ. وتعتبر المرجع الشامل لأي مُتحكِم.

سنتناول الموضوعات المختلفة في دليل البيانات على مدار فصول الكتاب بالكامل، حيث سنتعلم في كل فصل أحد الخصائص التي تتمتع بها مُتحكِمات AVR وسنحصل على تفاصيل هذه الخصائص من دليل البيانات.

هذا الفصل سيركز على الجزء الأول من دليل البيانات والذي غالباً ما يكون أول 5 أو 8 صفحات ويحتوي على النظرة العامة للمُتحكِم.

Features

- High-performance, Low-power Atmel® AVR® 8-bit Microcontroller
- Advanced RISC Architecture
  - 131 Powerful Instructions – Most Single-clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 16 MIPS Throughput at 16 MHz
  - On-chip 2-cycle Multiplier
- High Endurance Non-volatile Memory segments
  - 16 Kbytes of In-System Self-programmable Flash program memory
  - 512 Bytes EEPROM
  - 1 Kbyte Internal SRAM
  - Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
  - Data retention: 20 years at 85°C/100 years at 25°C[1]
  - Optional Boot Code Section with Independent Lock Bits
    In-System Programming by On-chip Boot Program
    True Read-While-Write Operation
  - Programming Lock for Software Security
- JTAG (IEEE std. 1149.1 Compliant) Interface
  - Boundary-scan Capabilities According to the JTAG Standard
  - Extensive On-chip Debug Support
  - Programming of Flash, EEPROM, Fuses, and Lock Bits through the JTAG Interface
- Peripheral Features
  - Two 8-bit Timer/Counters with Separate Prescalers and Compare Modes
  - One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode
  - Real Time Counter with Separate Oscillator
  - Four PWM Channels



8-bit AVR® Microcontroller with 16K Bytes In-System Programmable Flash

ATmega16
ATmega16L

الصفحة الأولى من دليل بيانات ATmega16

2.5 الخصائص العامة للمُتحكِم ATmega16/ATmega32

في الصفحة الأولى من دليل البيانات نجد المعلومات المتعلقة بالخصائص العامة للمُتحكِم (مع العلم أن هذه الخصائص تكون مشتركة بين معظم أفراد العائلة من المُتحكِمات وقد تختلف فيما بينها بفروقات بسيطة) وهي كالتالي:

المعمارية Advanced RISC (Harvard based) Architecture

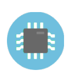
يوضح هذا الجزء الخصائص العامة لتقنية معالجة البيانات وسرعة المُتحكِم الدقيق حيث نجد أن المُتحكِم يتمتع بالقدرات التالية:

- **عدد 131 – single cycle execution Instruction** والتي تعني أن المُتحكِم يمكن برمجته باستخدام 131 أمر بلغة الأسمبلي ومعظم هذه الأوامر يتم تنفيذها في نبضة واحدة فقط.

- **16 Mega Instruction Per Second (MIPS)** وهي نفس الخاصية السابقة والتي تعني أن المُتحكِم يمكنه تنفيذ 16 مليون أمر برمجي عندما يتم تشغيله بتردد 16 ميجاهرتز (هذا بسبب أن معظم الأوامر البرمجية يمكنه تنفيذها في نبضة واحدة فقط). وتعبر هذه الخاصية عن أقصى سرعة معالجة للمُتحكِم الدقيق وتعتبر من أهم الخصائص التي تتمتع بها معالجات AVR.

> **مثال:** إذا كان لدينا برنامج مكون من 10 أوامر بلغة الأسمبلي والمتحكم يعمل بسرعة 1 ميجاهرتز (مما يعني أن زمن كل نبضة = **<u>1 ميكروثانية</u>**) فهذا يعني أن البرنامج سيستغرق تنفيذه زمن 10 نبضات وهو ما يساوي 10 ميكروثانية فقط.

- **On-Chip 2 cycle multiplier** في الأجيال القديمة من المعالجات والمُتحكِمات الدقيقة كان يتم حساب عملية ضرب الأرقام باستخدام الجمع المتكرر فمثلا حاصل ضرب 12×10= جمع رقم 12 مع نفسه 10 مرات (12+12+12+12+.....الخ). وهذا يعني تنفيذ أمر "الجمع"10 مرات (بلغة الأسمبلي) وبالتالي تستغرق وقت =10 نبضات. أما في مُتحكِم الـ AVR نجد وحدة معالجة الضرب تقوم بتنفيذ أي عملية ضرب في نبضتين فقط مما يسرع هذا النوع من العمليات الحسابية بصورة كبيرة.



الذاكرة عالية التحمل High endurance Memory

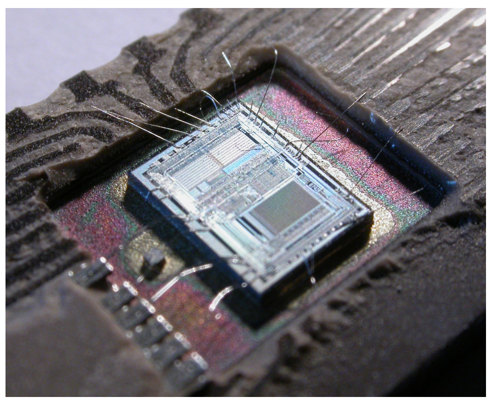
يوضح هذا الجزء الخصائص التي تمتاز بها الذاكرة الموجودة داخل مُتحكِمات AVR بأنواعها المختلفة مثل الـ Flash والـ RAM والـ EEPROM ومن أهم هذه الخصائص التالي:

- **16 كيلو بايت من الذاكرة الثابتة Self-programmable Flash memory** والتي تستخدم لحفظ البرنامج الذي سيشغل المُتحكِم الدقيق وتتمتع بخاصية البرمجة الذاتية التي تحدثنا عنها سابقاً. (يمتلك ATmega32 ذاكرة ثابتة 32 كيلوبايت).
- **512 بايت (512* 8 بت) من ذاكرة EEPROM**
- **1 كيلوبايت من الذاكرة المؤقتة (العشوائية) SRAM**
- **إمكانية الكتابة(برمجة)\مسح محتوى ذاكرة الفلاش نحو 10,000 مرة**
- **إمكانية الكتابة(برمجة)\مسح محتوى ذاكرة EEPROM نحو 100,000 مرة**.
- **الاحتفاظ بالبيانات في كل من الـ Flash والـ EEPROM لمدة زمنية تصل إلى 100 عام كامل** عند تشغيل المُتحكِم في درجة حرارة 25 درجة مئوية أو 25 عام عند تشغيل المُتحكِم في درجة حرارة 80 درجة مئوية وهذا يعني أن المُتحكِم يستطيع العمل والاحتفاظ بالبيانات لفترة طويلة جداً.
- **إمكانية استخدام Bootloader** (التفاصيل مذكورة في فصل الفيوزات)**.**
- **True Read-While-Write Operation** تعني أن المُتحكِم يستطيع قراءة بيانات من الـ ROM بينما يقوم بكتابة بيانات في الـ RAM في نفس الوقت على عكس المُتحكِمات القديمة والتي كانت تستطيع أن تقوم باحدى هذه العمليات فقط في نفس اللحظة.
- **Programming locks** طبقة من الحماية لمنع سرقة البيانات المخزنة على ذاكرة المُتحكِم باستخدام الـ Lockbits (التفصيل مذكورة في فصل الفيوزات).

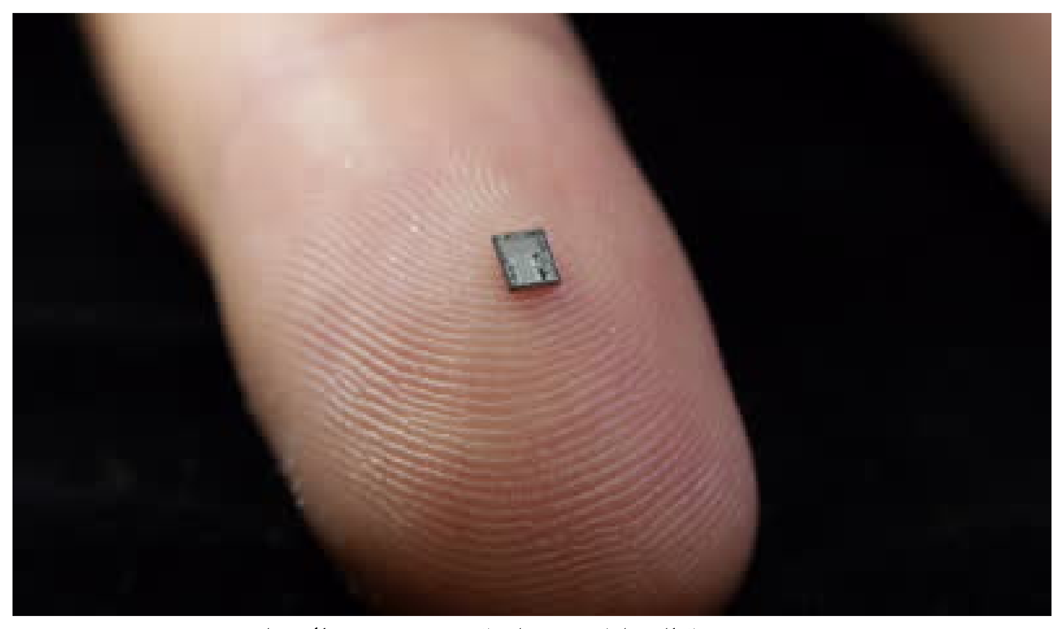
الأجزاء التالية من دليل البيانات مثل Jtag, Peripheral devices, Power consumption سيتم شرحها في فصولها الخاصة.

Microcontroller packaging

الحجم الحقيقي لشريحة السيليكون التي يتكون منها المُتحكِم الدقيق غالباً ما يكون صغير جداً لدرجة أنه قد يصل إلى حجم "رأس عود ثقاب" مما يجعل استخدامه مباشرة عملية صعبة، لذا يتم تصميم هيكل خارجي أكبر حجماً من مادة الـ Epoxy ويسمى الـ Packaging (الغلاف) للمُتحكِم الدقيق, ويخرج منه بعض الأطراف المعدنية الصغيرة التي تتصل بالمتحكم الدقيق الحقيقي.



صورة توضح المُتحكِم الدقيق من الداخل والذي لا يتخطى حجمه (في معظم الحالات) أكثر من 10% من حجم الغلاف

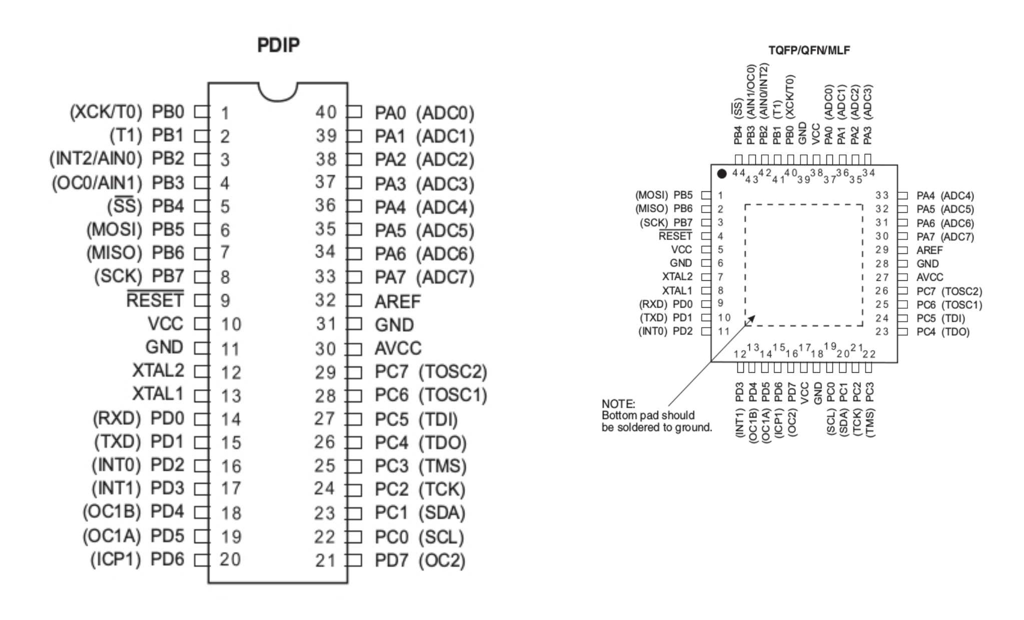


حجم شريحة السيليكون مقارنة بحجم إصبع الإنسان

تتوفر المُتحكِمات الدقيقة بكلا النوعين من التغليف DIP و SMD و سنجد أن شركة أتمل عادة ما تقوم بإصدار معظم المُتحكِمات من عائلة atmega و attiny بكلا النوعين DIP و SMD فمثلا سنجد في الصفحة الثانية من دليل البيانات وحدات الـ package الخاصة بالمتحكم ATmega16 وهي

- **DIP**
- **SMD – TQFP**

ملاحظة: ال SMD يتوفر منه أحجام وأشكال مختلفة مثل TQFP, BGA, QFN, MLF

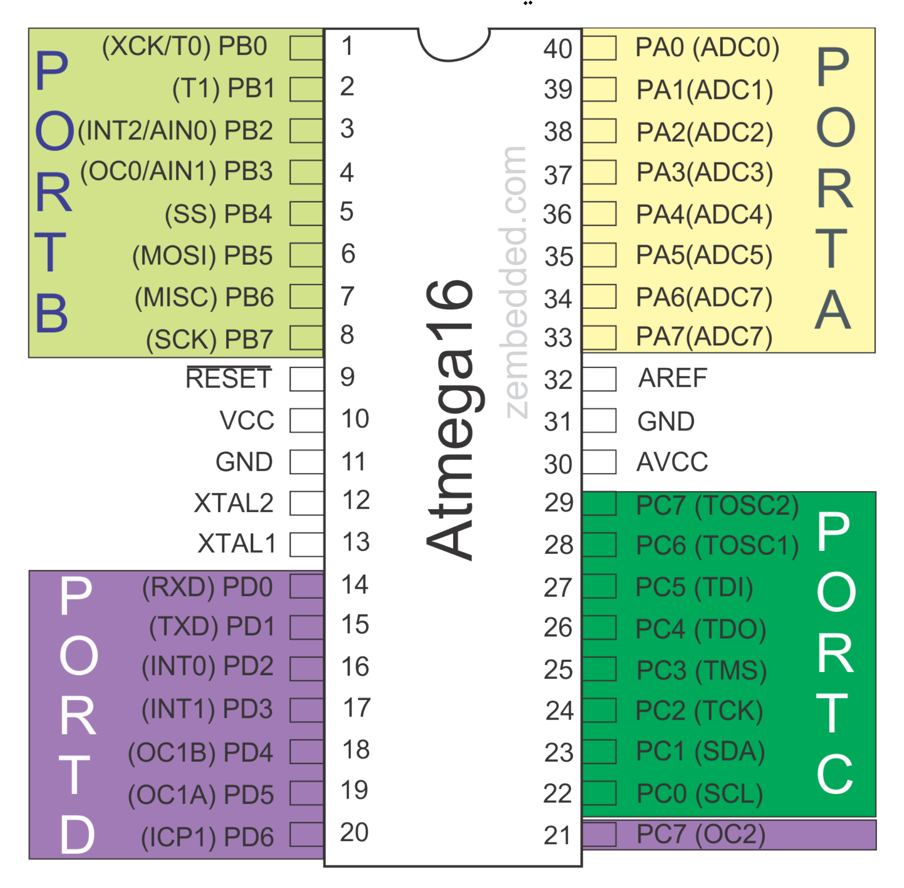


عندما تقوم الشركات بصناعة منتج ما فإنها تفضل استخدام معظم الشرائح الإلكترونية بتغليف SMD، حيث تتميز برخص السعر مقارنة بال DIP كما أنه يمكن تصميم دوائر تحتوي على الكثير من المكونات بمساحة صغيرة جداً بفضل الحجم الصغير الذي تتمتع به شرائح ال SMD.

ونجد نتيجة لهذا الأمر أن العائلات المتطورة مثل AVR32 أو Xmega غالباً ما يتم إنتاجها بتغليف SMD فقط وذلك لأنها تحتوي على الكثير من الأطراف (قد يصل إلى 120 طرف) لذا سيكون من الصعب (والمكلف أيضاً) أن تصنع بتغليف DIP لأن الحجم سيكون ضخم جداً.

2.6 أطراف المُتحكِم ATmega16

يتملك المُتحكِم ATmega16 ما مجموعة 40 طرف pin - موزعة على 4 بورتات كل واحد منهم 8 أطراف وهم PORTA, PORTB, PORTC, PORTD ومضاف إليهم مجموعة من الأطراف المتعلقة بالطاقة والتردد كما في الصورة التالية.



- **الطرف RESET** هذا الطرف يقوم بعمل RESET للمُتحكِم الدقيق ويعني أنه سيعيد تصفير جميع المُسجلات (يجعل قيمتها بصفر) ويعيد تشغيل البرنامج الموجود في ذاكرة المُتحكِم من البداية، مع العلم أنه طرف active low يعني يتم تفعيله عندما يتصل بالأرضي GND أو يحصل على إشارة LOW logic لذا يجب أن يوصل دائماً بال VCC عن طريق مقاومة 10 كيلو اوم (وإلا سيظل المُتحكِم يقوم بعمل RESET ولن يشغل البرنامج المُخزن بالذاكرة أبداً).

- **الطرف VCC** هذا الطرف الذي يستقبل الطرف الموجب للبطارية أو مصدر الطاقة المستخدم (يجب أن يكون موجب بداية من 2.7 فولت حتى 5.5 فولت بحد أقصى).

- **الطرف GND** الطرف الأرضي للمُتحكِم ويتم توصيله بالطرف الأرضي للبطارية أو مصدر الكهرباء المستخدم. قد تتساءل **لما يمتلك المُتحكِم زوج من الأطراف GND؟** السر هو تقليل الضجيج الكهربي Noise، فعندما يتواجد أكثر من مسار للأرضي فإن ذلك يحسن في القضاء على الـ Noise خاصة إذا كان المُتحكِم يقوم بتوليد إشارات عالية السرعة (بالميجاهرتز).

- **الطرفان XTAL1, XTAL2** الأطراف التي يتم توصيلها بدائرة المذبذب الخارجي والتي سنتعرف على جميع أنواعها بالتفصيل في فصل (الفيوزات، سرعة التشغيل والطاقة).

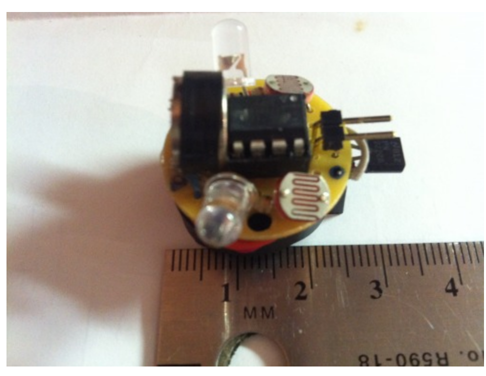
- **الطرف AVCC** اختصار لكلمة ADC VCC هذا الطرف مسؤول عن تشغيل المحول التناظري الرقمي ADC الموجود داخل المُتحكِم ويجب أن يتم توصيله دائماً بنفس الجهد الذي يتصل به الـ VCC.

- **الطرف AREF** اختصار كلمة Analog Refrence والذي سيتم شرحه بالتفصيل مع الـ ADC

باقي أطراف المُتحكِم موزعة على البورتات المختلفة A,B,C,D والتي تمتلك القدرة على التحكم بالمكونات الإلكترونية المختلفة كما تستطيع استقبال البيانات القادمة من الحساسات (سواء كانت رقمية أو تماثلية) لذا تسمى " منافذ إدخال\إخراج عامة" GPIO كما تمتلك مجموعة من الوظائف الإضافية Alternative functions مثل الاتصالات التسلسلية، المقاطعات .. الخ، والتي سنتعرف عليها تبعا في الفصول التالية.

2.7 عائلة ATTiny

تعتبر هذه العائلة من المُتحكِمات حديثة نسبياً وهي مخصصة للاستخدامات التي تحتاج مُتحكِم دقيق صغير الحجم وسريع في ذات الوقت دون التضحية بالمميزات الكاملة التي قد تجدها في معظم المُتحكِمات لذا قامت شركة ATmel بإنتاج هذا الجيل المتميز من المُتحكِمات والمسمى ATTiny اختصاراً لعبارة ATmel Tiny .



صورة روبوت صغير باستخدام المُتحكِم ATTiny85

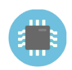
عندما ننظر للصفحة الأولى من دليل البيانات للمتحكمات ATTiny 45/84/85 نجد أن هذه المُتحكِمات تمتلك معظم القدرات الموجودة في سلسلة megaAVR فهي تمتلك نفس المعمارية وتعمل بسرعات تصل إلى 20 ميجاهرتز مع القدرة على تنفيذ 20 مليون أمر برمجي في الثانية الواحدة (هذا يعني أنها تتفوق على المُتحكِمات الأقدم نسبياً مثل ATmega16/32/128).

Features
- High Performance, Low Power AVR® 8-Bit Microcontroller
- Advanced RISC Architecture
  - 120 Powerful Instructions – Most Single Clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
- Non-Volatile Program and Data Memories
  - 2/4/8K Bytes of In-System Programmable Program Memory Flash
    - Endurance: 10,000 Write/Erase Cycles
  - 128/256/512 Bytes of In-System Programmable EEPROM
    - Endurance: 100,000 Write/Erase Cycles
  - 128/256/512 Bytes of Internal SRAM
  - Data Retention: 20 years at 85°C / 100 years at 25°C
  - Programming Lock for Self-Programming Flash & EEPROM Data Security
- Peripheral Features
  - One 8-Bit and One 16-Bit Timer/Counter with Two PWM Channels, Each
  - 10-bit ADC
    - 8 Single-Ended Channels
    - 12 Differential ADC Channel Pairs with Programmable Gain (1x / 20x)
  - Programmable Watchdog Timer with Separate On-chip Oscillator
  - On-chip Analog Comparator
  - Universal Serial Interface
- Special Microcontroller Features
  - debugWIRE On-chip Debug System
  - In-System Programmable via SPI Port
  - Internal and External Interrupt Sources: Pin Change Interrupt on 12 Pins
  - Low Power Idle, ADC Noise Reduction, Standby and Power-Down Modes

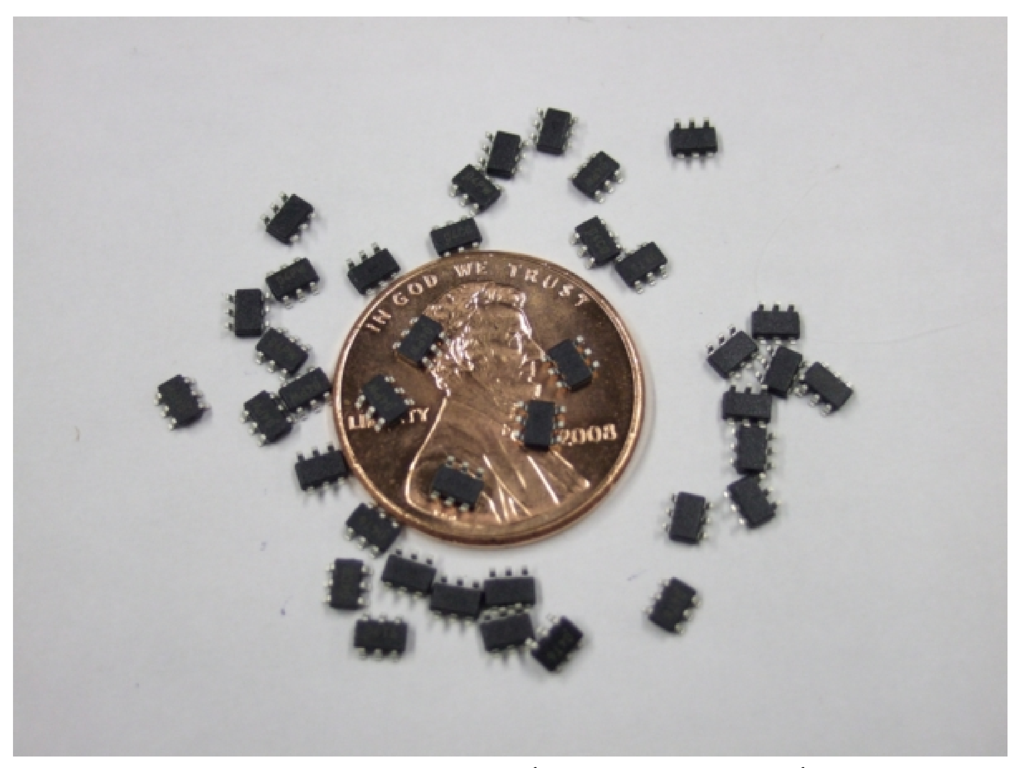


8-bit AVR® Microcontroller with 2/4/8K Bytes In-System Programmable Flash

ATtiny24
ATtiny44
ATtiny84

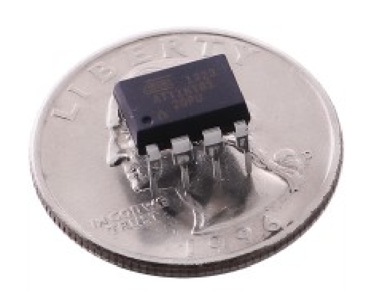


الفارق الأساسي بين هذه العائلة و megaAVR هو الحجم وعدد الأطراف التي تعمل كـ GPIO حيث نجد أصغر مُتحكِم في هذه العائلة يملك 6 أطراف فقط منهم 4 GPIO و 2 للطاقة مثل ATTiny4/5/9/10. **وتتميز مُتحكِمات هذه العائلة بالقدرة على العمل بفرق جهد 1.8 فولت.**



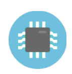
مجموعة من أصغر المُتحكِمات من شركة أتمل مقارنة بعملة معدنية و تسمى ATTiny10

بالرغم من الحجم الصغير جداً إلا أن هذه المُتحكِمات تتضمن معظم الـ Peripheral Devices الموجودة في مُتحكِمات megaAVR فمثلا المُتحكِم ATTiny85 بالرغم من امتلاكه 6 أطراف تحكم فقط إلا أنه يمكن تشغيلها كـ GPIO, ADC, PWM, SPI.



ATtiny45 / ATtiny85

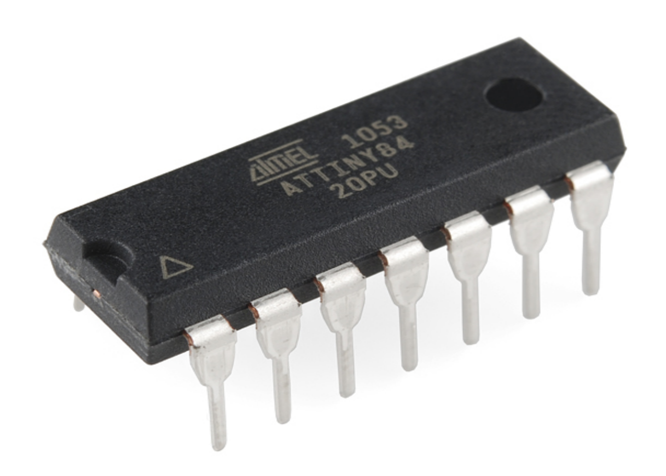
Reset	1	8	VCC (+)
(Analog Input 3) Pin 3	2	7	Pin 2 (Analog Input
(Analog Input 2) Pin 4	3	6	Pin 1 (PWM, MISO)
(–) GND	4	5	Pin 0 (PWM, AREF,



سنستخدم في التجارب القادمة المُتحكِم **ATtiny84** (بجانب المُتحكِم **ATmega16**) والذي يمتلك 12 طرف تحكم و 2 طرف للطاقة.

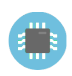
ATtiny44 / ATtiny84

(+) VCC	1	14	GND (–)
Pin 10	2	13	Pin 0 (Analog Input 0, AREF)
Pin 9	3	12	Pin 1 (Analog Input 1)
Reset	4	11	Pin 2 (Analog Input 2)
(PWM) Pin 8	5	10	Pin 3 (Analog Input 3)
(PWM, Analog Input 7) Pin 7	6	9	Pin 4 (Analog Input 4, SCK)
(MOSI, PWM, Analog Input 6) Pin 6	7	8	Pin 5 (Analog Input 5, PWM, MISO)



أيضاً تتميز هذه المُتحكِمات بالاستهلاك المنخفض جداً للطاقة. حيث يمكنها العمل على فارق جهد 1.8 فولت واستهلاك تيار يصل إلى 300 ميكروأمبير = 0.000300 أمبير (300 جزء من مليون من الأمبير) وهو ما يعني استهلاك أقل بنحو 12 مرة من استهلاك مُتحكِمات megaAVR والتي تستهلك تيار كهربي بمقدار 1 مللي أمبير (1000 ميكروأمبير) على الأقل.

إدارة استهلاك الطاقة مشروحه بالتفصيل في الفصل السابع

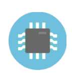


2.8 تمارين إضافية

- قارن بين الخصائص التي يتمتع بها المُتحكِم ATmega16 والمتحكم ATmega32 والمتحكم ATTiny84 من حيث الذاكرة وعدد أوامر البرمجة.
- كم عدد البورتات التي يملكها المُتحكِم ATTiny84 وما هي أسماؤها؟ هل جميع البورتات تمتلك 8 أطراف مثل المُتحكِم ATmega16 أم يوجد اختلاف ؟
- إذا قمنا بكتابة برنامج مكون من 100 أمر بلغة الأسمبلي بدون استخدام أي أمر تأخير وتم تشغيل نفس البرنامج على المُتحكِم ATmega16 والمتحكم ATTiny84، أي المتحكمين سيقوم بتنفيذ البرنامج أسرع ولماذا؟

> **نصيحة:** لمعرفة الحل انظر إلى فارق سرعة تنفيذ الأوامر MIPS في الصفحة الأولى من دليل البيانات لكل مُتحكِم ثم احسب زمن تشغيل البرنامج على كل مُتحكِم.

- إذا كان عدد الأوامر البرمجية لنظام تحكم يصل إلى خمسة كيلوبايت فما هو المُتحكِم المناسب لتشغيل هذا البرنامج (اختر ما يصلح من بين ATmega16, ATtiny84 ,attiny85 ,attiny45 ,ATmega32)؟ وإذا كان حجم البرنامج 17 كيلوبايت هل تصلح جميع اختياراتك السابقة؟



2.9 مراجع إضافية

مقارنة شاملة بين عائلة المُتحكِمات ATTiny

- www.microfusion.de/e-/Microcontroller/AVR-Overview/ATtiny.html
- en.wikipedia.org/wiki/Atmel_AVR_ATtiny_comparison_chart

مقارنة شاملة بين عائلة megaAVR

- ATmega32-avr.com/avr-comparison/

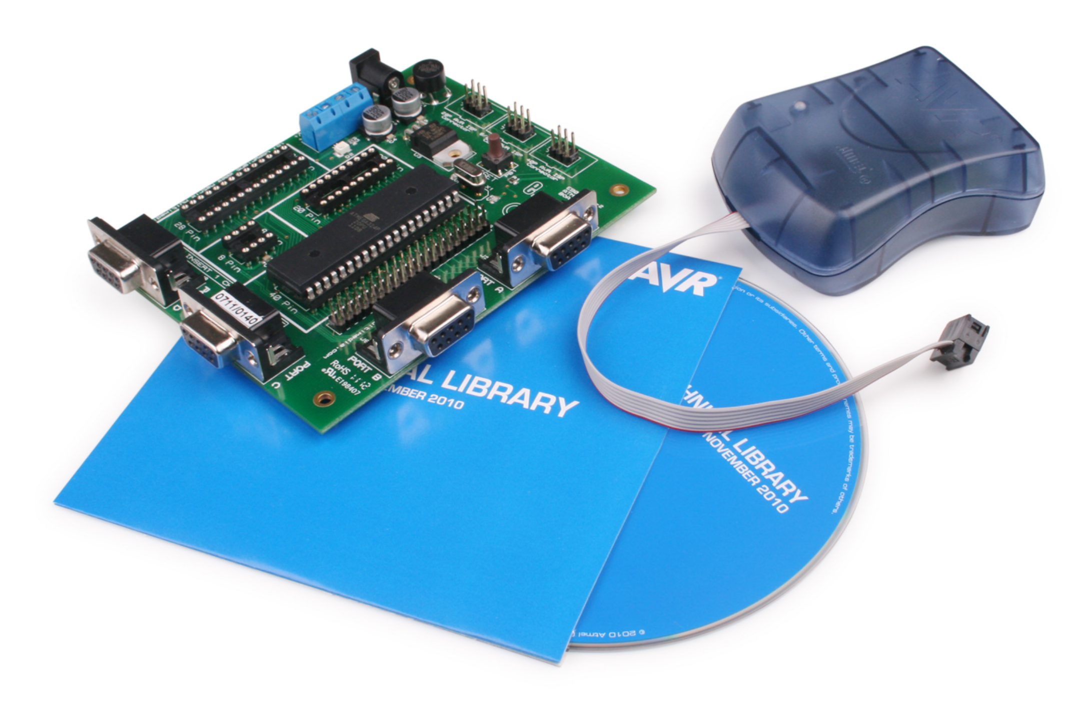
مرفق مع الكتاب ملف مقارنة شامل بين أفراد المُتحكِمات في كلا العائلتين ATTiny و megaAVR

الفصل الثالث

" هناك اختياران مبدئيان في الحياة: إما أن نتقبل ظروف الوضع الحالي كما هي، أو أن نتقبل مسؤولية تغيير هذا الوضع "

د. دينيس ويتلي - كاتب ومحاضر أمريكي

3. تجهيز أدوات التجارب



يوضح هذا الفصل الأدوات التي سنستخدمها في تطوير الأنظمة المدمجة سواء كانت العتاد "المكونات الإلكترونية" Hardware أو الأدوات البرمجية ToolChain (Softwares)

- ✔ المُبرمجات (الحارقات Burners).
- ✔ المكونات الإلكترونية
- ✔ البرمجيات المستخدم في التطوير Toolchain

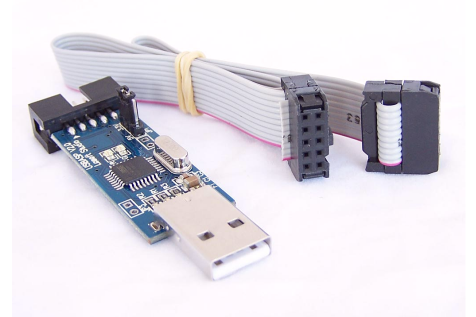
3.1 المُبرمجات

تعتبر المُبرمجات من أهم القطع الإلكترونية لتعلم المُتحكِمات الدقيقة فهي المسؤولة عن تحميل البرنامج الذي نكتبه على الحاسب إلى المُتحكِم الدقيق نفسه. البعض يحب تسميتها بالحارقات Burner نسبة إلى عملية حرق "الكود البرمجي" على المُتحكِم.
يمكن استخدام العديد من المُبرمجات المتوفرة في السوق ولكن هنا سأذكر لك أفضل هذه المُبرمجات وما يميزها ولك حرية الاختيار بينها (أي واحدة منهم ستفي بالغرض).

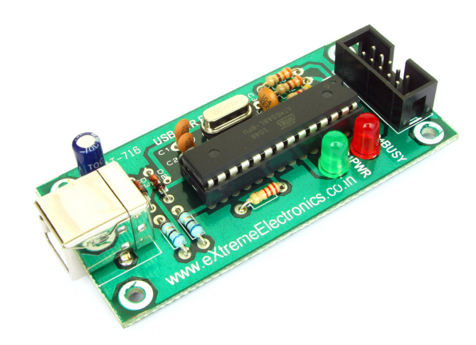
USBasp

تعتبر أحد أبسط وأشهر المُبرمجات التي تعمل عبر مدخل usb التقليدي وتتميز بالسعر المنخفض حيث يمكنك صناعتها بنفسك بتكلفة 3 دولارات أو شرائها جاهزة بسعر يتراوح بين 6 إلى 9 دولارات، كما أنها تحتوي على وضع الرفع البطيء slow clock rate mode وهو من الأوضاع المهمة في برمجة المُتحكِمات عندما تعمل على سرعات منخفضة (سنتحدث عن هذا الوضع بالتفصيل في فصل التلاعب بالترددات والطاقة) ، بهذا السعر المنخفض وسهولة البناء تعتبر USBasp أشهر مبرمجة الـ AVR على الإطلاق.

الصور التالية هي أشكال مختلفة من نفس المُبرمجة

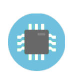


الإصدار الـ *SMD* من المُبرمجة *USBasp*



الإصدار الـ *DIP* من المُبرمجة *USBasp*

> الموقع الرسمي للمُبرمجة USBasp يحتوي على جميع ملفات التصميم التي يمكنك تحميلها مجاناً وصناعتها بنفسك www.fischl.de/usbasp



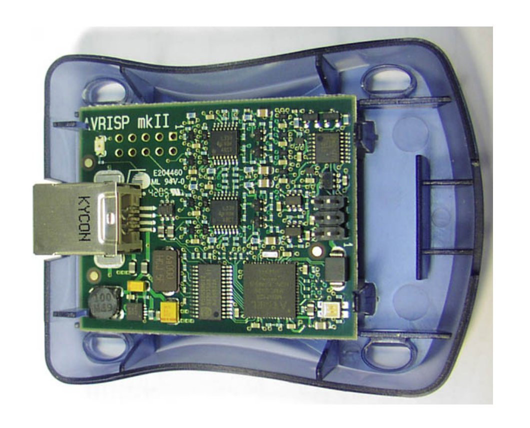
AVRISP mkII – Atmel

المُبرمجة الرسمية من شركة أتمل، وتعد من المُبرمجات المتطورة نسبياً حيث تدعم معظم عـائلات الـ AVR الـ 8 بـت والـ 32 بـت مثـل Xmega ويمكنهـا برمجـة المُتحكِمـات بمختلـف السرعات من 500 هرتز إلى 8 ميجا هرتز (هذه السرعة تمثل سرعة نقل البيانات وليس سرعة المُتحكِم نفسه) .

تعتبر AVRISP mkII أفضل من USBasp بكثير حيث تدعم USBasp سرعات برمجة بحد أقصى 5 ميجا فقط – كما أنها تدعم برمجة المُتحكِمات التي تعمل بفرق جهد من 1.8 فولت حتى 5 فولت. بالتأكيد كل هذه المميزات تأتي على حساب السعر الذي يبلغ نحو 40 دولار تقريباً. يمكنك قراءة كافة التفاصيل عنها من الرابط التالي
http://www.ATmel.com/tools/AVRISPMKII.aspx



المُبرمجة AVRISP mkII من الخارج



المُبرمجة AVRISP mkII من الداخل

تحذير: تنتشر بعض النسخ الصيني المزورة من هذه المُبرمجة بسعر منخفض (حوالي 10 دولارات – ولديها نفس الغطاء الخارجي) احترس من هذه المُبرمجات لأنها لا تحتوي على التصميم والمكونات الحقيقة للمبرمجة avrisp mkII وبالتالي لا تمتلك المميزات المذكورة سابقاً.

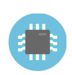
AVR Dragon

إذا كنت تريد أقوى أداة للتعامل مع عائلات AVR إذاً عليك بهذا "التنين" . تعتبر المُبرمجة AVR Dragon أقوى أداة للتعامل من مُتحكِمات AVR فهي تعمل كمبرمجة SPI أو OCD أو PDI وكمنقح Jtag Debugger ويمكنها معالجة الفيوزات والبرمجة بالفولتية العالية High voltage burner – 12 Volt ويمكنها صيانة المُتحكِمات المعطوبة أو نسخ ومعالجة المحتوى المكتوب على المُتحكِمات (سيتم شرح الفيوزات بالتفصيل في الفصل الخاص بها).

يمكنك الاطلاع على تفاصيل هذه المُبرمجة من موقع Atmel الرسمي من الرابط التالي:
www.ATmel.com/webdoc/avrdragon



تتوفر هذه المُبرمجة بسعر 55 دولار أمريكي، وتعتبر من الأدوات المُخصصة للمحترفين لما تملكه من إمكانيات متطورة.



المُبرمجات ذات التغطية العامة Universal Programmers

هذا النوع من المُبرمجات يمكنه التعامل مع جميع المُتحكِمات الدقيقة وشرائح الذاكرة من مختلف الشركات فمثلاً معظم هذه المُبرمجات يمكنها برمجة PIC, AVR, ARM, 8051, EPROM والمزيد من الشرائح والمُتحكِمات. غالباً تستخدمها شركات الصيانة والتطوير لأنها توفر الجهد وتوفر شراء العديد من المُبرمجات لمختلف الأنواع.





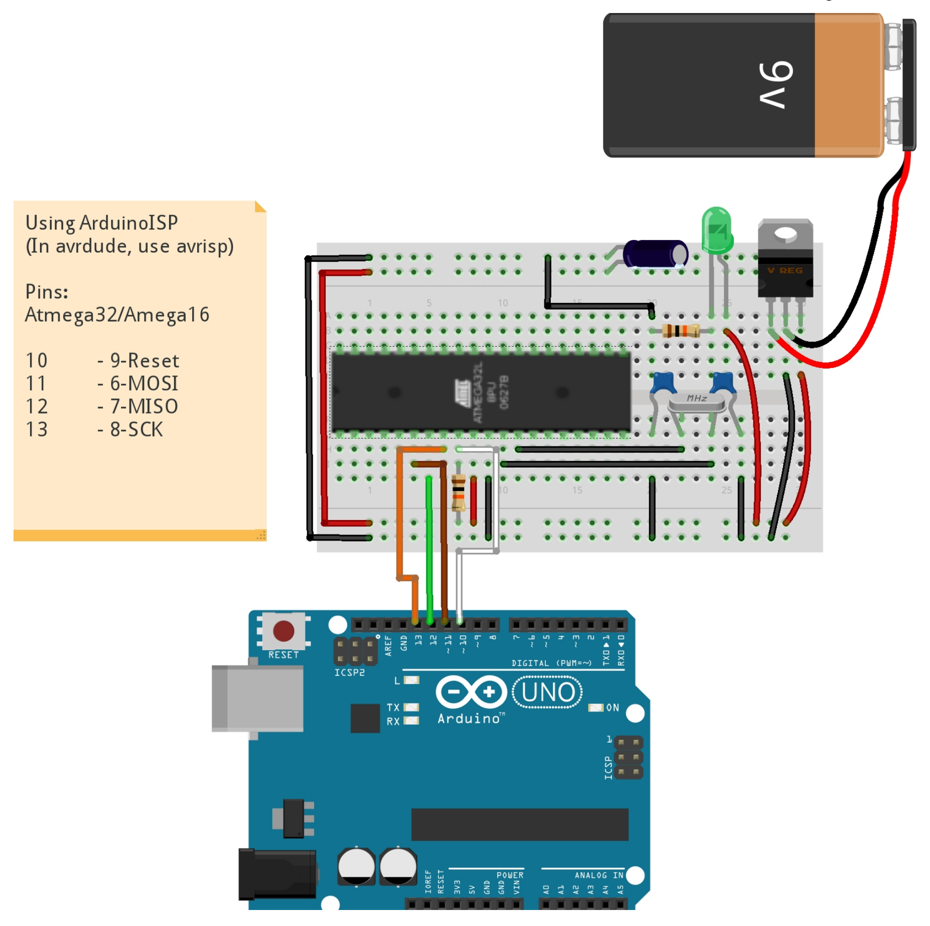
المشكلة الوحيدة لهذا النوع هو سعرها المرتفع جداً والذي يبدأ من 100 دولار وحتى 1200 دولار (قد يبدو رقم 1200 دولار ضخم لكن لك أن تتخيل أن هذه المُبرمجات يمكنها التعامل مع أكثر من 8000 شريحة إلكترونية من مختلف الشركات على هذا الكوكب).

> لا أنصحك بشراء هذا النوع إلا إذا كنت ترغب في تعلم استخدام أكثر من نوع من المُتحكِمات الدقيقة ولا تريد أن تتعب نفسك بشراء مبرمجات مختلفة.

Arduino as ISP

قد يستغرب البعض لما تم وضع آردوينو في قائمة المُبرمجات، الحقيقة أن لوحات آردوينو المختلفة يمكنها العمل كمنصة تطوير AVR لأنها من الأساس عبارة عن شريحة Atmega328 مضاف إليها بعض المكونات البسيطة + مُحول USB-ttl converter. هذه اللوحات يمكنها برمجة AVR بطريقتين:

الطريقة الأولى: أن تقوم برفع برنامج يسمى ArduioISP على لوحة آردوينو والذي سيقوم بتحويل اللوحة إلى مبرمجة مشابهه لـ USBasp ويمكنك بعدها أن توصلها بأي شريحة AVR لتبرمجها كما في الصورة التالية والتي يتم استخدام لوحة آردوينو بها لبرمجة شريحة ATmega32 أو ATmega16



الطريقة الثانية: أن تستخدم لوحة آردوينو نفسها كـ AVR development board حيث يمكنك الكتابة بلغة السي ANSI- C داخل برنامج آردوينو.

في الواقع معيار آردوينو البرمجي ما هو إلا لغة السي ومضاف إليها بعض المكتبات البرمجية لذا يمكنك بسهولة أن تكتب أي برنامج ANSI - C داخل برنامج آردوينو.

الخبر الجيد أنه في حالة امتلاكك لوحة آردوينو فلا داعي لشراء أي مبرمجة إضافية ويكفي فقط أن تشتري باقي المكونات المطلوبة وتستغل لوحة آردوينو لتقوم بهذا العمل.

الروابط التالية تشرح استخدام آردوينو كمبرمجة (مثل الطريقة الأولى):

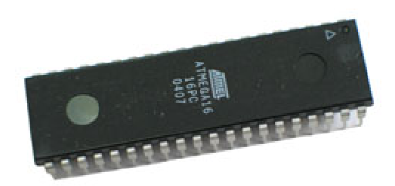
- www.youtube.com/watch?v=_ZL-YNOH_jA
- www.arduino.cc/en/Tutorial/ArduinoISP
- www.instructables.com/id/AVR-Programming-with-Arduino-AVRdude-and-AVR-gcc/

شخصياً أستخدم المُبرمجة USBasp في أغلب الأوقات وتعتبر المُبرمجة المفضلة لدي بسبب سعرها المنخفض وسهولة صناعتها في المنزل.

في هذا الكتاب سنعتمد على شريحة ATmega16 و شريحة ATTiny لذا فالطريقة الثانية من استخدام آردوينو كلوحة تطوير AVR وبالتحديد المُتحكِم atmega328 قد تحتاج أن تعدل بعض الأكواد البرمجية المذكورة في الكتاب (خاصة الأكواد المذكورة بدئاً من الفصل السابع حتى نهاية الكتاب) وذلك بسبب اختلاف أسماء بعض المُسجلات Register وأسماء بعض المخارج والمداخل، لذا إذا رغبت في استخدام آردوينو كـ AVR board فأحرص على تغير أسماء المُسجِلات لتناسب الأمثلة.
أيضاً سيتم شرح الـ Datasheet لكل مُتحكِم. والتي من خلالها يمكنك تغيير هذه الأسماء بسهولة و تطبيق كافة الأكواد مع تعديلها قليلاً لتتناسب مع المُتحكِم Atmega328 بدلاً من المُتحكِمات المذكورة سابقاً.

3.2 المكونات الإلكترونية

ستحتاج لبعض المكونات الإلكترونية لأداء التجارب في هذا الكتاب، بعض هذه المكونات واجب توافره والبعض الآخر يُستخدَم فقط في التجارب الإضافية لذا احرص على اقتناء المكونات التي سيكتب بجانبها **(واجب)** - أما المكونات التي سيكتب بجانبها **(اختياري)** فيمكنك الاستغناء عنها (ومع ذلك أنصحك بشرائها حتى تستفيد بأكبر قدر من التجارب).



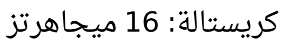
شريحة ATmega16

شريحة ATTiny84

(واجب) هذا هو المُتحكِم الرئيسي الذي سنقوم بعمل التجارب عليه، وفي حالة عدم توافره لديك في السوق المحلي يمكنك استخدام البديل المماثل ATmega32 والذي يماثله في معظم التركيب الداخلي باستثناء مساحة الذاكرة.

(اختياري) المُتحكِم ATTiny84





كريستالة: 16 ميجاهرتز

(واجب) عدد 1 كريستالة (ذات طرفين) 16 ميجا



مكثف سيراميكي 22 بيكوفاراد picofarad

(واجب) عدد 2 مكثف على الأقل.

هذه المكثفات سعرها رخيص جداً وحجمها صغير وسهلة الضياع لذا يفضل أ ن تشتري منها 10 أو 20 قطعة (الـ 20 قطعة ستكلفك نصف دولار فقط).

كابل مبرمجة ISP



(واجب) عدد 1 كابل

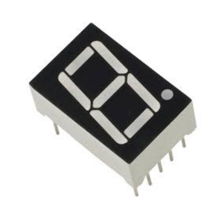
يُستخدَم هذا الكابل في توصيل المُبرمجة بالمتحكم الدقيق (غالباً قد تجده مع المُبرمجة نفسها عندئذ لا داعي لشراءه).



دايودات ضوئية (ليد LED)

(واجب) عدد 10 قطع دايود ضوئي (يفضل اللون الأحمر أو الأخضر) ويفضل عدم استخدام اللون الأبيض

شريحة عرض أرقام "مقاطعة سباعية" seven Segment

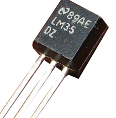


(واجب) عدد: 2 قطعة

من المكونات الهامة والبسيطة في ذات الوقت، تستخدم في عرض الأرقام عن طريق 7 دايودات ضوئية لذا تسمى الشريحة ذات المقاطع السبعة 7 Segment ويفضل الحصول على النوع ذا الطرف السالب المشترك common cathode

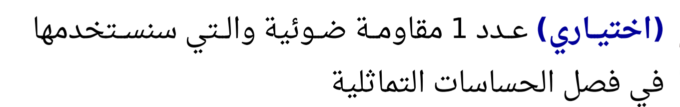
---

حساس الحرارة LM35



(اختياري) عدد 1 حساس حرارة والذي سنستخدمه في فصل الحساسات التماثلية

مقاومة ضوئية LDR



(اختياري) عدد 1 مقاومة ضوئية والتي سنستخدمها في فصل الحساسات التماثلية

مقاومة متغيرة potentiometer



(واجب) عدد 1 مقاومة متغيرة ذات 3 أطراف توصيل مع عمود دوران

محرك DC Motor



(واجب) عدد 1 محرك تيار مستمر مثل الموجود في الألعاب مثل السيارات

محرك Stepper Motor (bipolar)



(اختياري) عدد 1 محرك خطوي من نوع bipolar له سلك اذا 4 أطراف ويستخدم في تقنيات التحكم للماكينات، يجب أن لا يستهلك تيار أكبر من 500 مللي أمبير (نصف أمبير) وسنتعرف عليه بالتفصيل في فصل المحركات.

دائرة قيادة L293 H-bridge



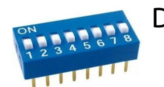
(اختياري) دائرة قيادة المحركات (قنطرة H) وتعتبر من أهم الشرائح التي سنستخدمها في التحكم بالمحركات سواء الـ DC أو الـ Stepper Motor

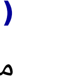


مفتاح Push button

(واجب) عدد 4 مفاتيح ضغط ذات أربعة أطراف،إذا لم تكن متوفرة في السوق المحلية يمكنك شراء مفاتيح الضغط ذات الطرفين فقط.

مفتاح DIP switch



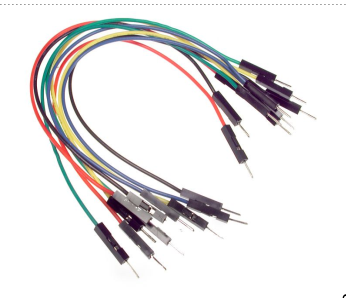


(اختياري) عدد 1 مصفوفة ON- OFF – DIP ذا ثمانية مفاتيح أو يمكنك شراء 2 مصفوفة ذات 4 مفاتيح

Resistors
10 كيلواوم
330 اوم

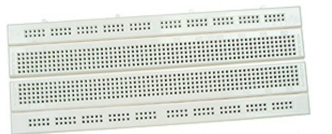



(واجب) عدد 8 مقاومات من كل القيم المذكورة (10 كيلواوم و 330 اوم) بقدرة ربع واط أو: watt 1/8



أسلاك توصيل

(واجب) عدد 20 سلك يمكنك شراء الأسلاك الخاصة بلوحة التجارب مباشرة Male to Male Jumpers أو يمكنك أن تصنعها بنفسك عبر شراء سلك (خط تليفون أرضي 0.6 مللي) وتقطيعها بنفسك، شخصياً أفضل استخدام الأسلاك الجاهزة، ويفضل أن تكون بطول ما بين 10 إلى 15 سنتي متر.



لوحة تجارب Breadboard

(واجب) عدد 1 لوحة تجارب Breadboard، احرص على اقتناء لوحة كبيرة الحجم ومن النوع الجيد لأنها ستكون المنصة التي ستطبق عليها مختلف التجارب.

3.3 أدوات إضافية

مـن المحبـذ أن تمتلـك مجموعـة مـن الأدوات الإضـافية حيـث ستسـاعدك كـثيراً فـي تعلـم المُتحكِمات الدقيقة والإلكترونيات بشكل عام لذا احرص على اقتنائها إذا لم تكن لديك بالفعل.
القائمة التالية هي مثال على ما ينبغي أن تمتلكه:

- جهاز قياس متعدد Digital Multimeter (AVO meter)
- عدة اللحام (كاوية + حامل للكاوية + حامل لوحات إلكترونية PCB holder)
- قصدير لحام من نوع 70-30
- مقص أسلاك (قصافه) + قشارة أسلاك.
- عدسة مكبرة
- مجموعة من المفكات + ماسكة أسلاك (بنسه).



3.4 تجهيز البرمجيات

البرمجيات المخصصة للأنظمة المدمجة (أو كما تسمى Firmware فيرم-وير) يتم تصميمها باستخدام "مجموعة أدوات التطوير" أو كما تعرف باسم Development toolcahin من الرائع أن هذه الأدوات متوفرة لمتحكمات الـ AVR مجاناً وبصورة مفتوحة المصدر لجميع أنظمة التشغيل، هذا يعني أن جميع البرمجيات التي سنستخدمها مجانية تماماً ويمكنك استخدامها بحرية بأي صورة سواء كانت تعليمية أو تجارية. مجموعة الأدوات هي:

- **المترجم AVR-GCC Compiler** – البرنامج الشهير GCC يعتبر أشهر مترجم في العالم وهو البرنامج الرئيسي في أدوات التطوير و المسؤول عن تحويل اللغات عالية المستوى مثل C/C++ إلى ملفات تستطيع الآلات أن تفهمها(بالتعاون مع برامج الـ Linker و الـ Assembler) قامت شركة ATmel بتعديل الـ GCC ليعمل مع الـ AVR مباشرة بمختلف الإصدارات AVR32-GCC ,AVR-GCC ,AVR-GCC-G++ في هذا الكتاب سنستخدم AVR-GCC فقط.
- **الأدوات المساعدة binutils** – مجموعة من الأدوات التي تساعد المترجم في إتمام عملية تحويل البرنامج من اللغة عالية المستوى إلى ملف hex متكامل (الهيكس hex هي صيغة نصية تمثل الـ binary في صورة أبسط وأكثر اختصاراً).
- **المكتبات البرمجية (LibC-avr) Libraries** – مجموعة من الأكواد والتعريفات المكتوبة مسبقاً بلغات عالية أو أقل مستوى لتسهل عملية البرمجة على المطورين وتحتوي على بعض الأكواد والأوامر الجاهزة والتي تختصر الكثير من وقت كتابة البرامج.
- **المنقح GDP debugger** – هذا البرنامج يُستخدَم في اكتشاف الأخطاء البرمجية والمساعدة في حل المشاكل التي تواجه المطور أثناء تشغيل واختبار البرنامج.
- **برنامج الرفع AVRdude** – البرنامج المستخدم في رفع الملفات من الحاسوب إلى المُتحكِمات الدقيقة من نوع AVR ويمكنه أيضاً أن يقوم بعكس هذه العملية (استخراج البرامج المكتوبة على المُتحكِمات) كما يمكنه قراءة محتويات الذاكرة EEPROM وكتابة الفيوزات (كما سنرى في الفصل الخاص بالفيوزات).

هناك طريقتان لاستخدام هذه الأدوات، الأولى هي تحميل الـ toolchain ثم استخدام أي IDE أو حتى محرر نصوص يدعم لغة السي \ السي++ والطريقة الثانية أن يتم تحميل منصة التطوير المتكاملة من شركة أتمل ATmel Studio – هذا البرنامج يحتوي على كل الأدوات السابقة مدمجة بداخله **(باستثناء AVRdude فقط).**

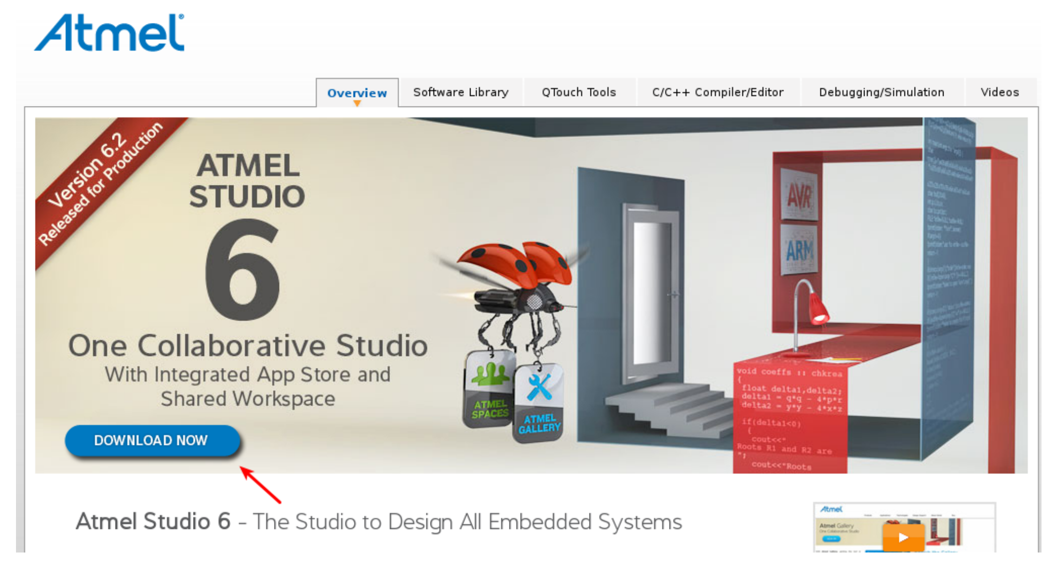
الطريقة الأولى تصلح لجميع أنظمة التشغيل ومناسبة جداً لأنظمة Linux و Mac أما الثانية مع الأسف برنامج ATmel Studio متوفر على نظام ويندوز فقط، على أي حال سأقوم بشرح كلا الطريقتين حتى يصبح لك حرية اختيار البرامج وحرية استخدام أي نظام تشغيل تريده.

أيضاً يمكنك استخدام برنامج المحاكاة الشهير بروتس Protues في محاكاة جميع الأمثلة المذكورة في فصول الكتاب دون الحاجة لشراء مكونات إلكترونية حقيقة. مع العلم أن برنامج بروتس يمكنه العمل على نظام لينكس أيضاً عبر استخدام محاكي ويندوز Wine HQ

معلومة إضافية: برنامج Atmel Studio يعمل بنفس ال toolchain المذكورة سابقاً ومنها المترجم AVR-GCC كذلك نجد أن برنامج آردوينو Arduino يعمل بنفس ال toolchain، وهذا يعني أنه يمكنك استخدام برنامج آردوينو كـ IDE خفيفة لتكتب برامج بلغة السي\السي++ لبرمجة ال AVR ولكني لا أحبذ هذا الخيار لأنه يفتقر للكثير من الوظائف الاحترافية والهامة لمبرمجي الأنظمة المدمجة.

تجهيز الأدوات على نظام ويندوز

توجه إلى موقع شركة Atmel لتحميل برنامج Atmel Studio من الرابط التالي
http://www.ATmel.com/tools/atmelstudio.aspx



اختر نسخة البرنامج المتكاملة والمضاف إليها باقة أدوات الـ NET. (هذه النسخة تحتوي على كل الملفات المطلوبة) وعادة ما تتميز بحجمها الكبير لذا احرص على أن يكون لديك اتصال سريع بالإنترنت عند تحميل البرنامج.



سيطلب منك الموقع أن تسجل حساب جديد (أو تسجل دخول إذا كان لديك حساب بالفعل)، الحساب مجاني تماماً وسيوفر لك تحميل جميع أدوات الشركة مجاناً. بعد الانتهاء من تسجيل الحساب يمكنك تحميل البرنامج.

بعد الانتهاء من تنصيب Atmel studio سنقوم بتحميل برنامج AVRdudess وهو برنامج avrdude مضاف إليه واجهة رسومية رائعة ومزودة بالعديد من الخيارات التي تُسهل برمجة شرائح الـ AVR، يمكنك تحميل البرنامج من الموقع التالي:

http://blog.zakkemble.co.uk/avrdudess-a-gui-for-avrdude



by Zak Kemble

AVRDUDESS is a GUI for AVRDUDE, a tool for programming Atmel microcontrollers.

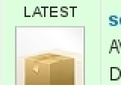
Some key features:

- Supports all programmers and MCUs that AVRDUDE supports
- Supports presets, allowing you to change between devices and configurations quickly and easily
- Drag and drop files for easy uploading
- Automatically lists available COM ports
- Cross-platform with the use of Mono for Linux & Mac OS X



Downloads

LATEST



setup-AVRDUDESS-2.4.exe (914.73 kB)
AVRDUDESS (Windows installer)
Downloaded 12370 times

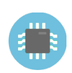
تنصيب تعريفات المُبرمجة USBasp

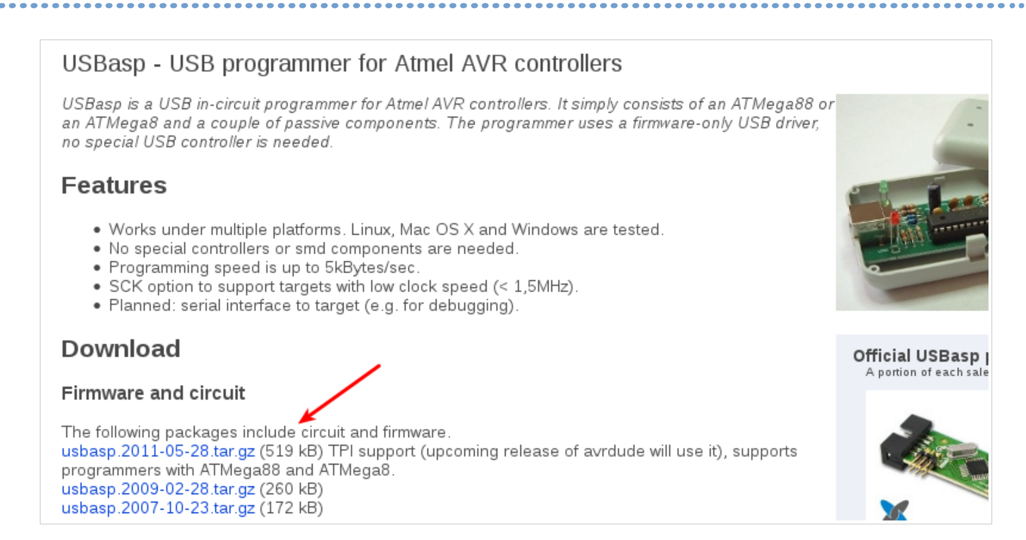
هذا الجزء قد يختلف على حسب المُبرمجة التي ستستخدمها، شخصياً أستخدم USBasp لأنها;كما ذكرت سابقاً رخيصة ومفتوحة المصدر ويمكنك صناعتها بنفسك. لذا اخترتها كأداة رئيسية للبرمجة في هذا الكتاب.

ملاحظة: إذا كنت تستخدم أحد مبرمجات ATmel مثل AVRISP أو AVR Dragon يمكنك تحميل التعريفات الخاصة بها من نفس صفحة برنامج ATmel studio

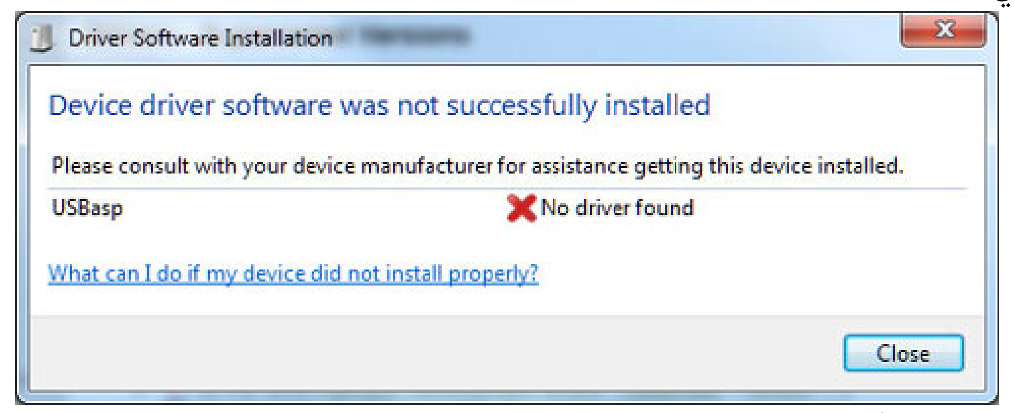
في البداية توجه إلى موقع USBasp الرسمي وقم بتحميل الملف المضغوط الذي يحتوي على جميع ملفات المشروع (ملفات التصميم والتعريفات)

http://www.fischl.de/usbasp

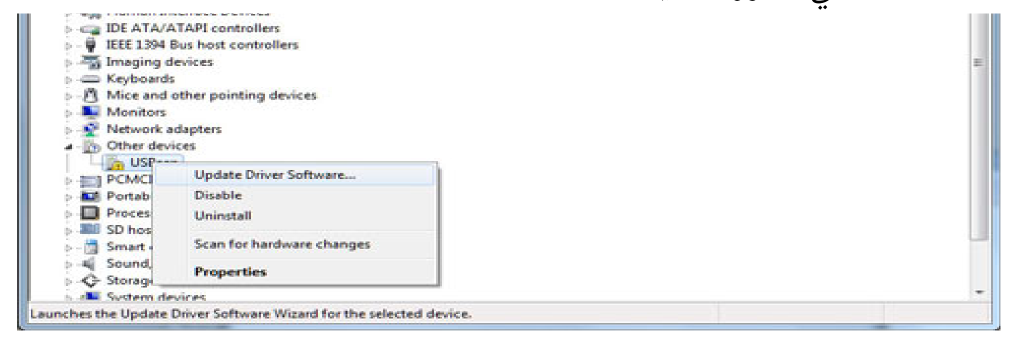


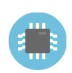


بعد الانتهاء من التحميل قم بتوصيل لوحة USBasp بالحاسوب وقم بفك ضغط الملف، لاحظ أنه بمجرد توصيل USBasp سيخبرك نظام ويندوز بأنه لم يستطع أن ينصب التعريفات كما في الصورة التالية:

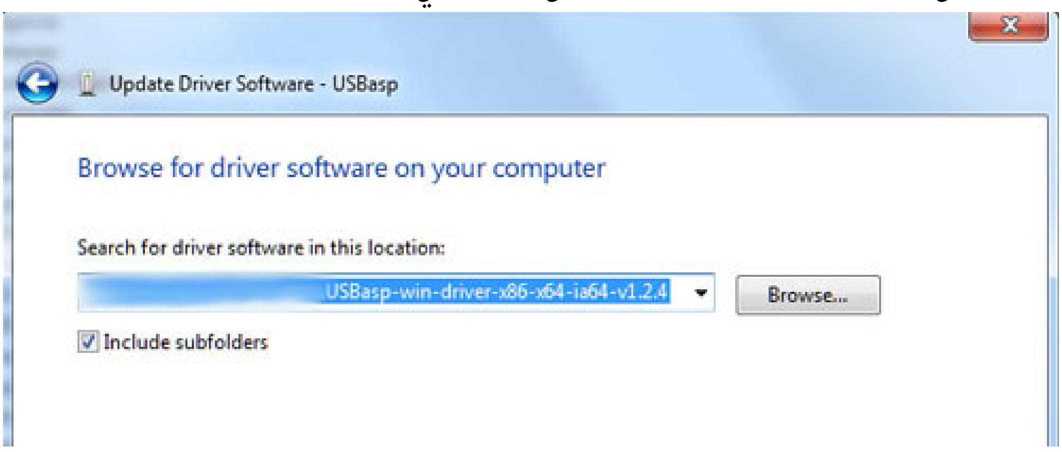


افتح مدير الأجهزة Device Manager واختر USBasp - Uknowen device ثم اضغط install new driver كما في الصورة التالية:





اختر مكان ملفات التعريفات (الموجودة داخل الملف الذي قمنا بتحميله سابقاً)



إذا ظهرت رسالة تطلب تأكيد تنصيب التعريفات قم بالضغط على Install

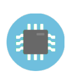


رائع ! أنت الآن جاهز لتبدأ العمل على صناعة الأنظمة المدمجة بال AVR :)

إذا كنت تستخدم نظام windows 8.1 أو Windows 10 يجب أن تقوم بإغلاق (نظام التعريفات الموثقة)، يمكنك قراءة التفاصيل من الرابط التالي

http://www.atadiat.com/usbasp_win_driver/

https://openchrysalis.wordpress.com/2014/09/26/installing-usbasp-driver-software-in-windows-8-1/



تجهيز الأدوات على أنظمة لينكس

في أنظمة لينكس \ ماك \ FreeBSD يمكنك تنزيل جميع برامج الـ toolchain من مستودعات البرامج الرسمية لكل نظام تشغيل، كما ستحتاج لملف MakeFile والذي سيقوم بتحويل الكود إلى ملف hex بصورة تلقائية (ملف الـ MakeFile مرفق مع الكتاب).

أيضاً ستحتاج IDE أو محرر نصوص يدعم البرمجة مثل البرنامج الرائع CodeBlocks (والذي أستخدمه شخصياً) أو Eclips أو Sublime أو المحرر النصي Geany أو Vim أو Emacs.

نظام Ubuntu / Debian

يمكنك تنصيب جميع الأدوات مباشرة عبر الأمر التالي (تكتب في طرفية سطر الأوامر Terminal)

```
sudo apt-get install gcc-avr binutils-avr gdb-avr avr-libc avrdude
```

نظام Fedora / RedHat / CentOS

يمكنك استخدام برنامج yum أو DNF (الموجود في نظام فيدورا لينكس 22 أو أعلى) وذلك عبر الأوامر التالية

```
sudo yum install avr-gcc avr-binutils avr-libc avr-gdb avrdude
```

لنظام فيدورا 22 أو أعلى

```
sudo dnf install avr-gcc avr-binutils avr-libc avr-gdb avrdude
```

تنصيب تعريفات USBasp على لينكس

يحتوي الملف الرسمي على تعريفات نظام لينكس (متوافقة مع جميع الأنظمة بلا استثناء) وهي عبارة عن ملف udev rules وسكربت تنصيب، كل ما عليك فعله أن تفتح المجلد الذي يحتوي على التعريفات وتشغل سكربت التنصيب بصلاحية الرووت كما في الصورة التالية:

```
[redacted]@localhost ~]$ cd "Downloads/usbasp.2011-05-28/bin/linux-nonroot"
[redacted]@localhost linux-nonroot]$ ls
99-USBasp.rules  install_rule
[redacted]@localhost linux-nonroot]$ sudo ./install_rule
```

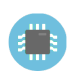
3.5 مراجع إضافية

- http://www.ladyada.net/learn/avr/programmers.html
- http://avrprogrammers.com/programmers/all
- http://www.instructables.com/id/AVR-ISP-programmer
- http://www.instructables.com/id/Turn-Your-Arduino-Into-an-ISP
- http://elm-chan.org/works/avrx/report_e.html
- http://www.instructables.com/id/Programming-an-Atmel-AtTiny85-using-Arduino-IDE-an/

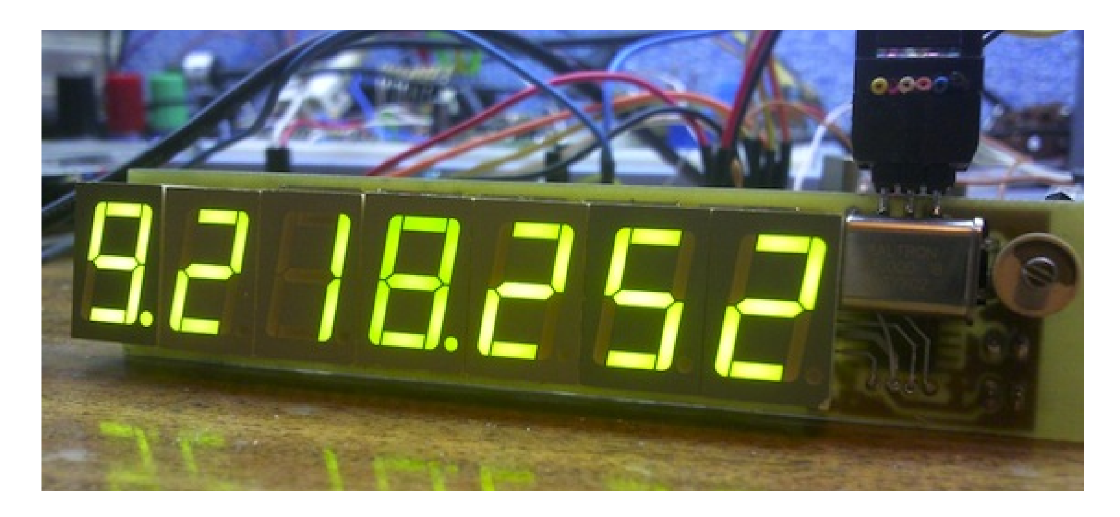
الفصل الرابع

" إن الخطر الذي يهدد الكثيرين منا ليس أن نضع أهدافاً عالية جداً فلا نستطيع بلوغها، بل أن نضع أهدافاً منخفضة للغاية، ثم نبلغها "

مايكل أنجلو - رسام ونحات إيطالي



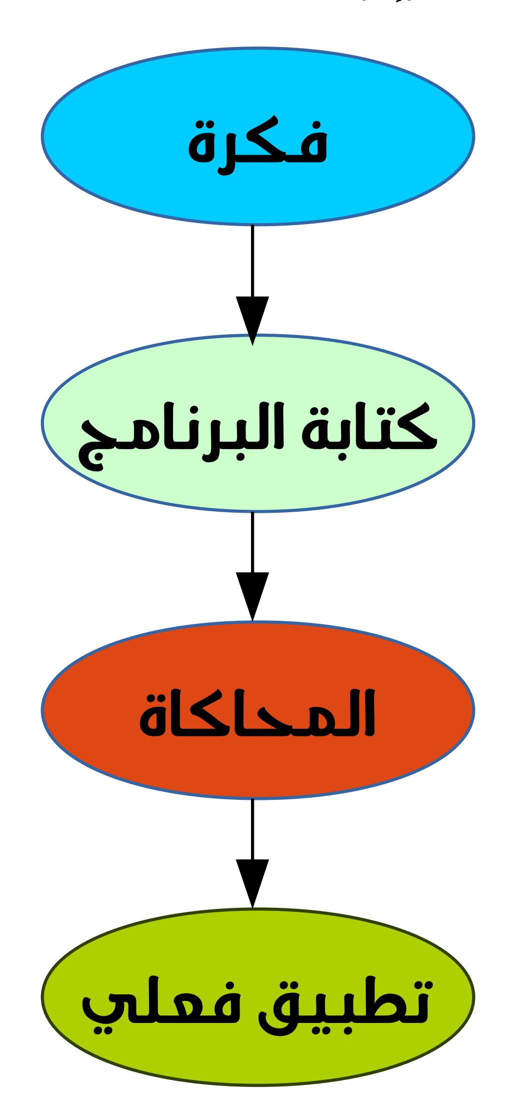
4. أساسيات التحكم GPIO Basics



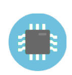
في هذا الفصل سنتعلم أساسيات تشغيل أطراف المُتحكِم الدقيق وتشغيل المنافذ لتعمل كدخل أو كخرج. كما سنقوم بمجموعة من التجارب لتشغيل بعض العناصر الإلكترونية البسيطة مثل LEDs, Switchs, 7-Segments ..الخ.

- ✔ المثال الأول: Hello World
- ✔ أساسيات برمجة أطراف الـ AVR
- ✔ المثال الثاني: تشغيل مجموعة دايودات ضوئية
- ✔ المثال الثالث: تشغيل جميع أطراف البورت A والبورت B
- ✔ المثال الرابع: تشغيل المقاطعة السباعية 7Segment
- ✔ قراءة الدخل الرقمي
- ✔ Internal & Externel Pull-Up
- ✔ المثال السادس: قراءة أكثر من مفتاح
- ✔ مفهوم الـ Bouncing وطرق الـ De-Bouncing

في جميع التجارب التالية سنتبع أسلوب التصميم Design ثم المحاكاة Simulate ثم التنفيذ على لوحة التجارب Prototype وذلك لتسهيل تعلم البرمجة، مع ملاحظة أنه هناك بعض الأمثلة التي قد لا تصلح للمحاكاة ويجب أن تنفذ مباشرة على لوحة التجارب كما سنرى في الفصول المتقدمة (مثل الـ fuses وإدارة الطاقة).



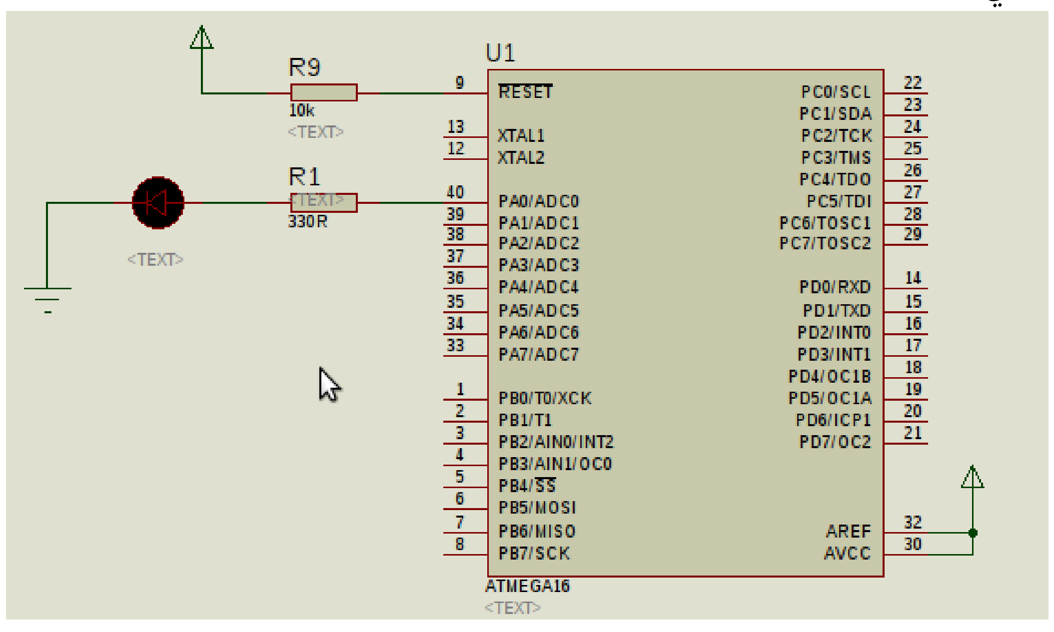
ملاحظة: جميع التجارب على لوحة الاختبارات Breadboard (التطبيق الفعلي) + العديد من التجارب الاضافية ستشرح على هيئة فيديوهات مستقله عن الكتاب



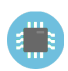
4.1 المثال الأول: Hello World

يعتبر تشغيل دايود ضوئي وإطفاؤه لمدة زمنية معينة Blinking Led هو المثال الأشهر لبدء أي عملية تطوير في عالم الأنظمة المدمجة. في هذا المثال سنتعرف على أساسيات التحكم في أطراف الـ AVR microcontrollers وتشغيلها كـ GPIO (منافذ إدخال وإخراج عامة).

سنستخدم في هذا المثال دايود ضوئي Led يتم توصيله على الطرف PA0 ويمكنك استخدام إما ATmega16 أو ATtiny84 (كلاهما يمتلكان الطرف PA0) كما هو موضح في المخطط التالي:

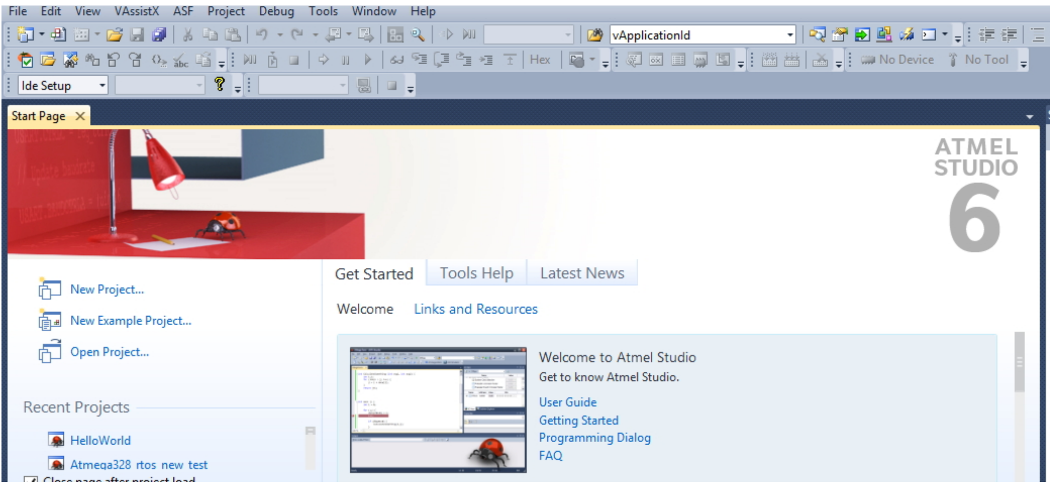


ملاحظة: كلمة مُخطط Schematic تعني الرسمة التي تشرح توصيل المكونات الإلكترونية بعضها البعض، دائماً ما يتم استخدام المخططات لشرح أي تصميم إلكتروني سواء كان بسيطاً أو مُعقداً. في هذا الكتاب سأستخدم برنامج بروتس لرسم معظم المخططات للدوائر التي سنقوم بتجربتها على مدار الأمثلة القادمة.

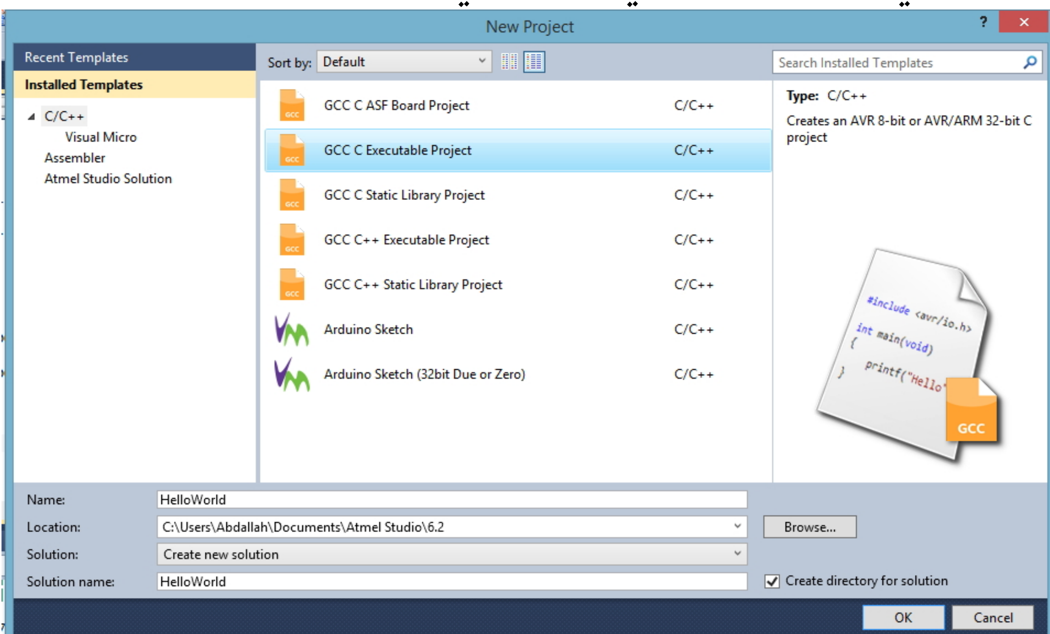


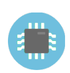
كتابة البرنامج على Atmel Studio

قم بتشغيل برنامج Atmel stduio ثم اختر New Project

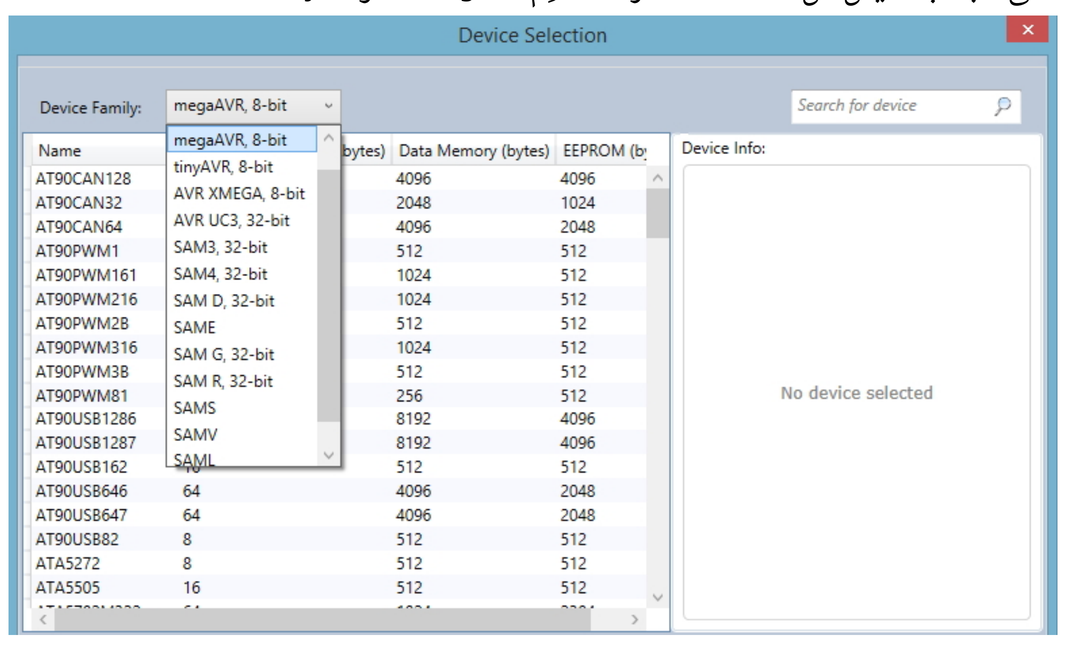


من الصفحة التي ستظهر اختر مشروع جديد بلغة السي GCC – C- Executable وقم باختيار اسم المشروع والمجلد الذي سيحفظ به المشروع من الشريط السفلي. **تذكر هذا المجلد جيداً لأنه سيحتوي على ملف الهيكس الذي سنستعمله في الخطوات التالية.**

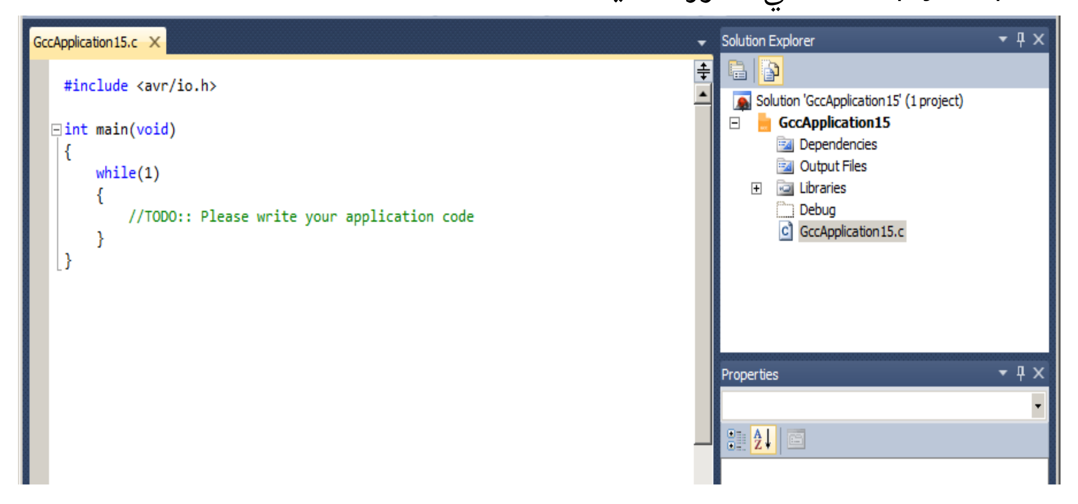


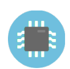


الآن يمكنك اختيار عائلة ونوع المُتحكِم الدقيق المستخدم، كما يمكنك استخدام مربع البحث على الجانب الأيمن من الصفحة (اختر المُتحكِم ATmega16 أو ATtiny84).



بعد الانتهاء من هذه الاختيارات ستظهر شاشة البرمجة الرئيسيةوبداخلها "هيكل فارغ" Empty template كما في الصورة التالية





والآن قم بكتابة أول برنامج

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <avr/delay.h>

int main(void)
{
   DDRA = 0b00000001;
   while(1)
   {
          PORTA = 0b00000001;
         _delay_ms(500);
          PORTA = 0b00000000;
         _delay_ms(500);
   }
   return 0;
}
```

شكل الكود بعد كتابه داخل البرنامج

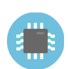
File Edit View VAssistX ASF Project Build Debug Tools Window Help

Debug | vApplicationId | Hex | Ide Setup

HelloWorld.c | ASF Wizard

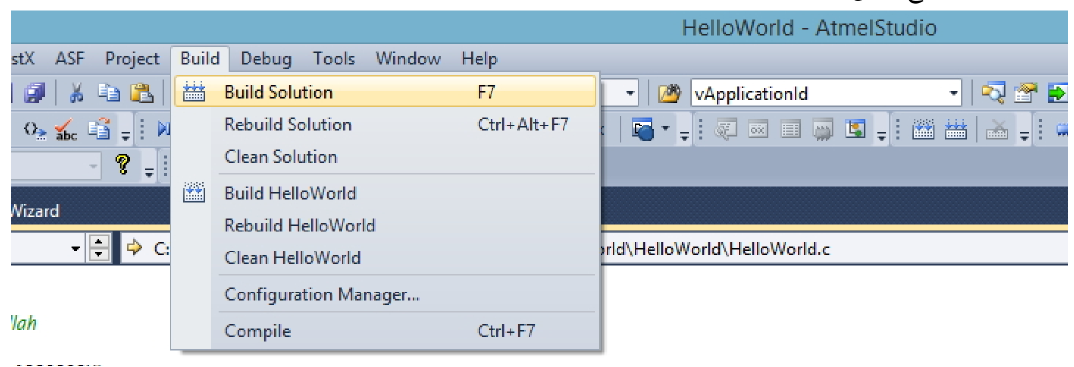
HelloWorld.c | C:\Users\Abdallah\Documents\Atmel Studio\6.2\HelloWorld\HelloWorld\HelloWorld.c

```
/*
 * HelloWorld.c
 *  Author: Abdallah
 */
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
int main(void)
{
    DDRA = 0b00000001;
    while(1)
    {
        PORTA = 0b00000001;
        _delay_ms(500);
        PORTA = 0b00000000;
        _delay_ms(500);
    }
    return 0;
}
```

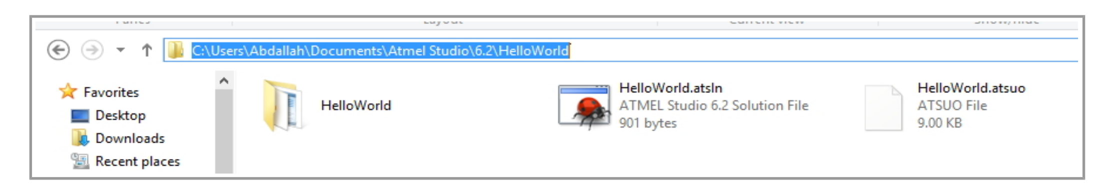


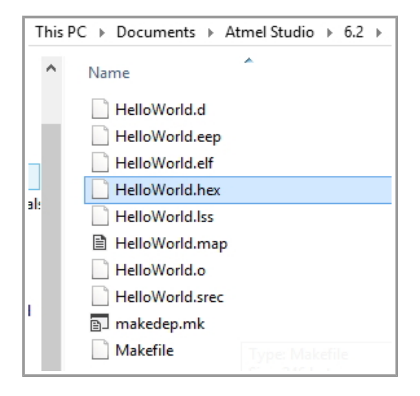
ترجمة الكود

لترجمة البرنامج وتحويلة من لغة السي إلى ملف الهيكس يمكنك الضغط على زر F7 أو اختيار "بناء البرنامج" من قائمة Build → Build **Solution**

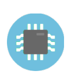


بعد الانتهاء من ترجمة البرنامج ستجد ملف الهيكس في المجلد الذي اخترناه في الخطوة الأولى، الآن يمكننا البدء في محاكاة التجربة على برنامج بروتس أورفع ملف الهيكس على المُتحكِم الدقيق مباشرة (على الـ Breadboard)





يمكنك استخدام طريقة تحويل الملفات باستخدام المترجم GCC مباشرة دون استخدام برنامج Atmel studio مثل ما هو مشروح في **ملحق استخدام makefile**

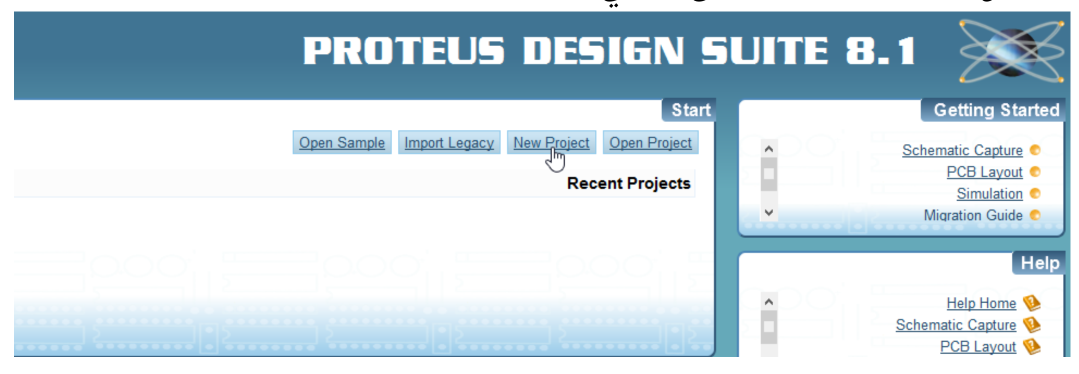


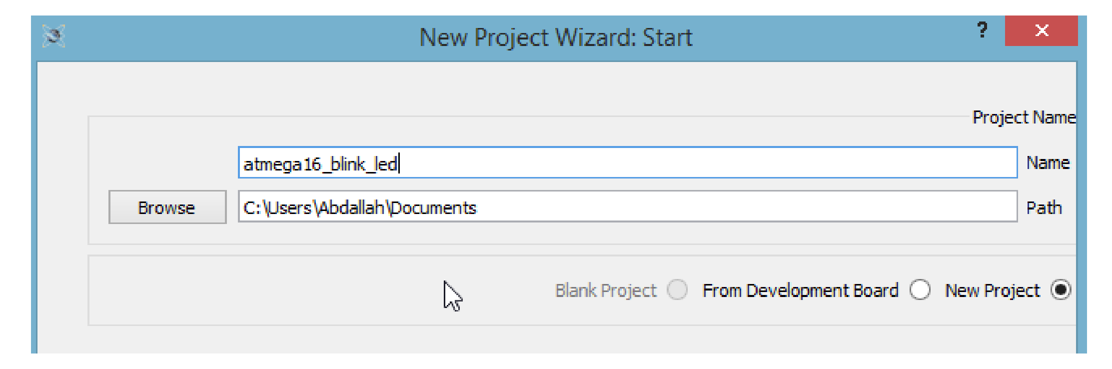
محاكاة التجربة على برنامج بروتس

والآن سنقوم بمحاكاة التجربة على برنامج بروتس الشهير والذي يعد أفضل برنامج محاكاة في مجال الإلكترونيات خاصة مع المُتحكِمات الدقيقة. البرنامج مخصص للعمل على نظام تشغيل ويندوز ومع ذلك يمكنك تشغيله على أنظمة **لينكس** بسهولة باستخدام محاكي برامج ويندوز **Wine (وهو ما أفعله شخصياً لأنني أفضل استخدام نظام لينكس).**

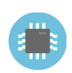
> يمكنك استخدام أي إصدارة من برنامج بروتس سواء كانت 7.8 SP2 أو الإصدار 8.1 مع العلم أن جميع الملفات المرفقة مع الكتاب تم تصميمها واختبارها على كلا الإصدارين

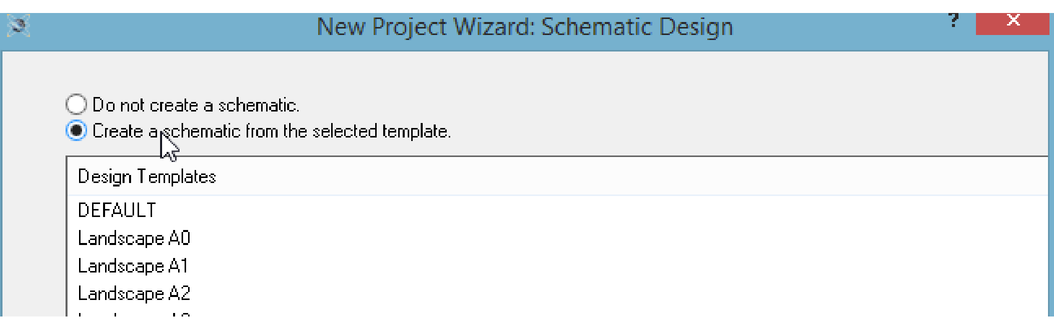
في البداية قم بعمل مشروع جديد ولنسميه ATmega16_blink_led واختر المكان الذي تريد أن تحفظ به ملفات المشروع كما في الصور التالية:





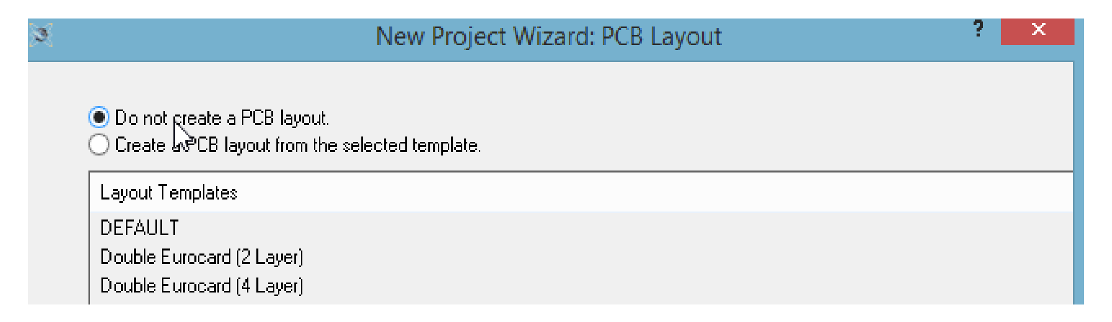
اختر تصميم مخطط جديد Create a new schematic بالمقاس الافتراضي Default





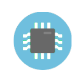
لن نحتاج أن نصنع تصميم PCB في الوقت الحالي، لذا قم باختيار Don't Create PCB ثم اختر في الصفحة التي تليها No Firmware

ملاحظة: الإصدارات الخاصة ببرنامج بروتس بدء من 8 أو أعلى تدعم برمجة المُتحكِمات من داخل البرنامج باستخدام المترجم AVRASM أو gcc-avr لكننا لن نستخدم هذه الخاصية الآن وسنكتفي باستخدام برنامج Atmel studio أو CodeBlocks كبيئة برمجة مع gcc-avr



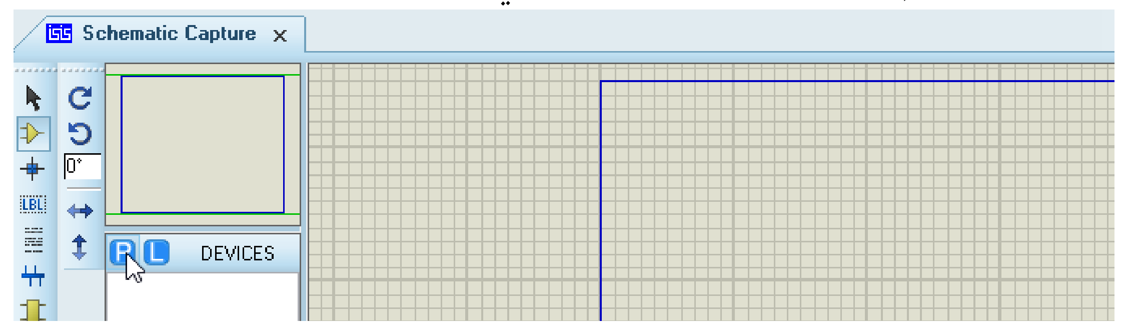


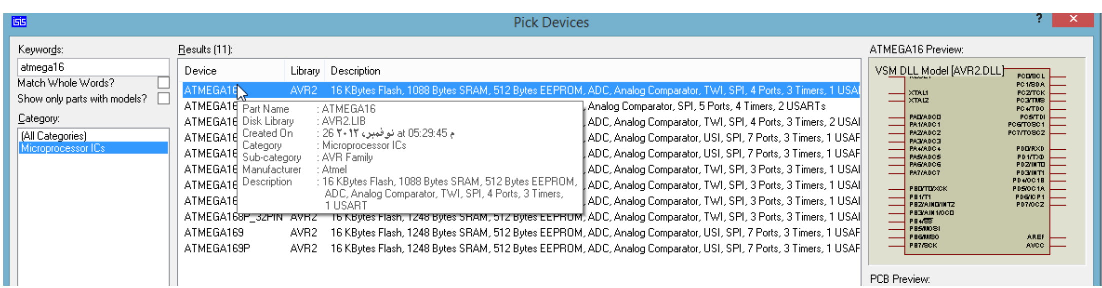
والآن أصبح لدينا ملف المخطط Schematic جاهز لنبدأ بتوصيل المكونات الإلكترونية مع بعضها البعض، في التجارب القادمة سنقوم باستخدام المكونات التالية (سنقوم بإضافتها في قائمة المكونات المستخدمة في المحاكاة من خلال الضغط على زر **P** من قائمة devices



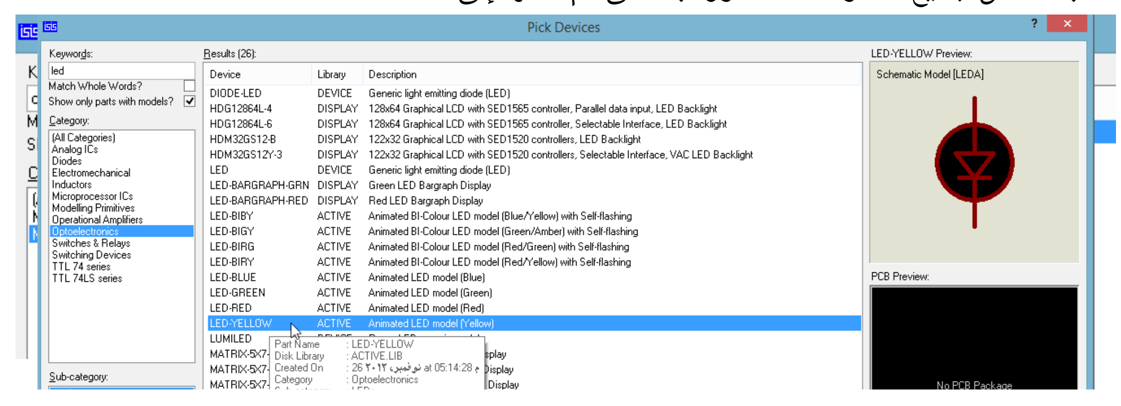
- ATmega16
- LED (yellow)
- Resistor 330
- LED bar

يمكنك استخدام خاصية البحث عن المكونات كما في الصور التالية

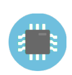


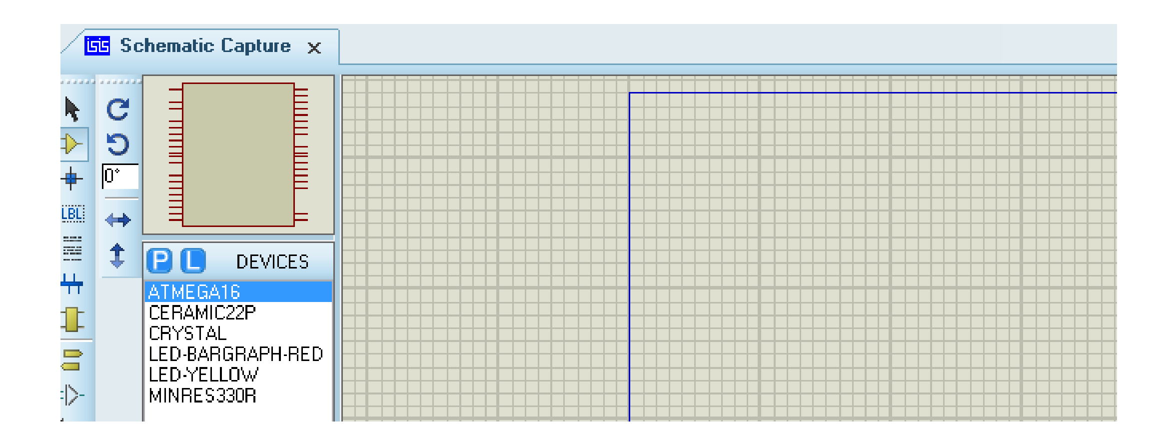


ابحث عن جميع المكونات المذكورة بالأعلى ثم ضفها إلى القائمة

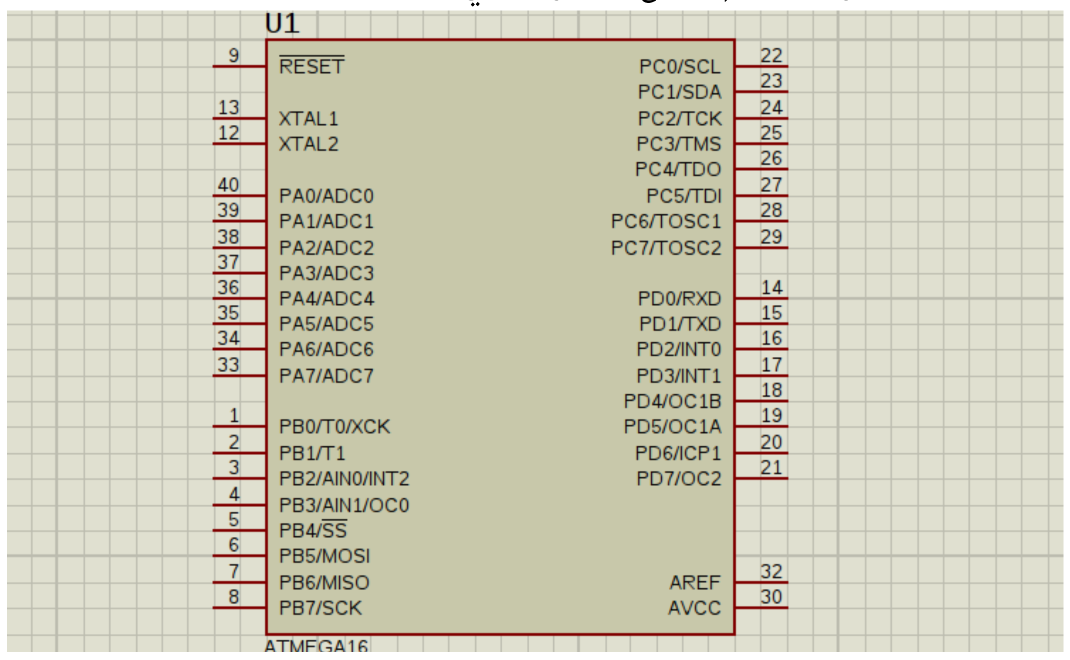


بعد إضافة جميع المكونات سنجد أن المستطيل الجانبي الخاص بالمكونات أصبح يحتوي على معظم المكونات التي نحتاجها للتجارب القادمة كما في الصورة التالية:

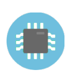


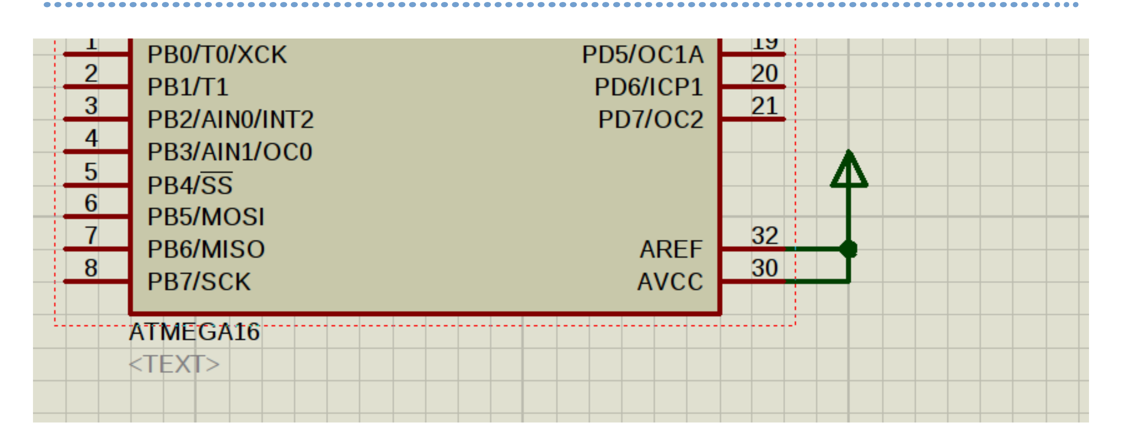


والآن سنبدأ في بناء أول دائرة لتجربة الـ Blinking led، في البداية سنقوم بوضع المُتحكِم ATmega16 داخل إطار رسم برنامج بروتس كما في الصورة التالية:

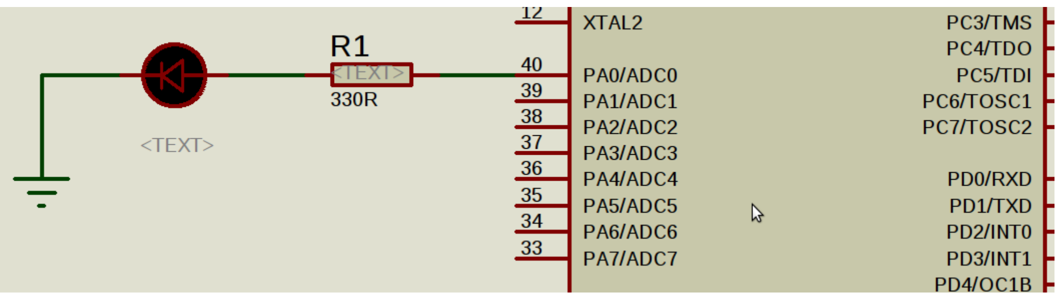


ثم سنختار قائمة terminal من برنامج بروتس (القائمة التي تحتوي على رموز البطارية – الموجب والسالب) ومنها سنضيف power ونقوم بتوصيلها بالمدخل AREF و AVCC (سنتحدث عن كلا المخرجين بالتفصيل في الفصل الخاص بالمحول التناظري\الرقمي ADC).

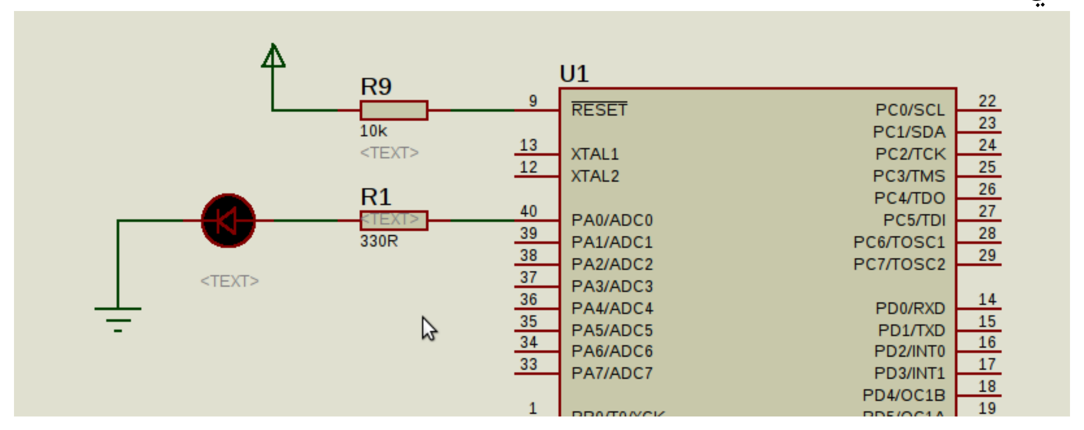


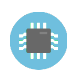


والآن قم بإضافة الدايود الضوئي والمقاومة على المخرج PA0 ثم وصل الطرف السالب من الدايود على علامة Ground (يمكنك إضافتها من قائمة Terminals أيضاً) كما في الصورة التالية:

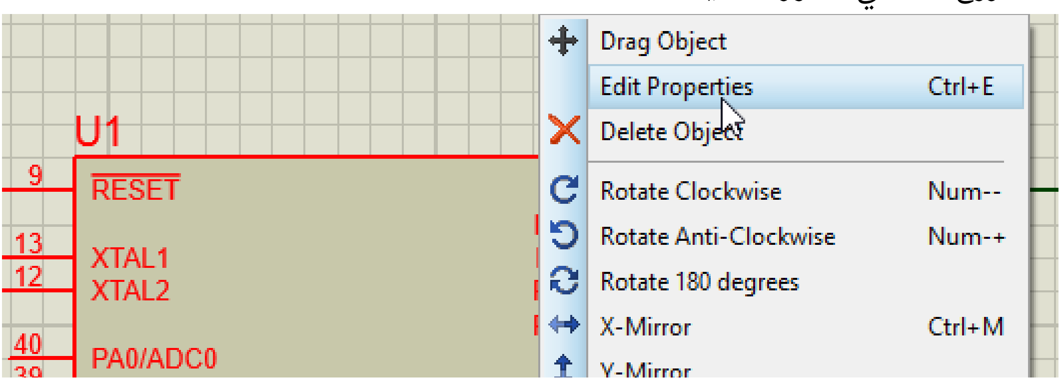


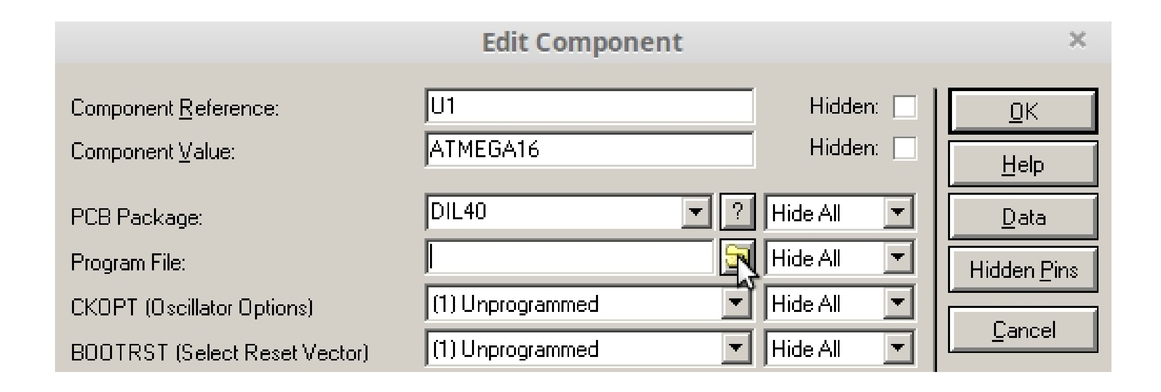
قم بإضافة مقاومة أخرى ووصلها بالمدخل RESET في الطرف الأيسر من المُتحكِم ثم وصل الطرف الآخر بعلامة Power – وقم بتعديل قيمة المقاومة لتصبح 10 كيلو (تكتب 10k) كما في الصورة التالية:

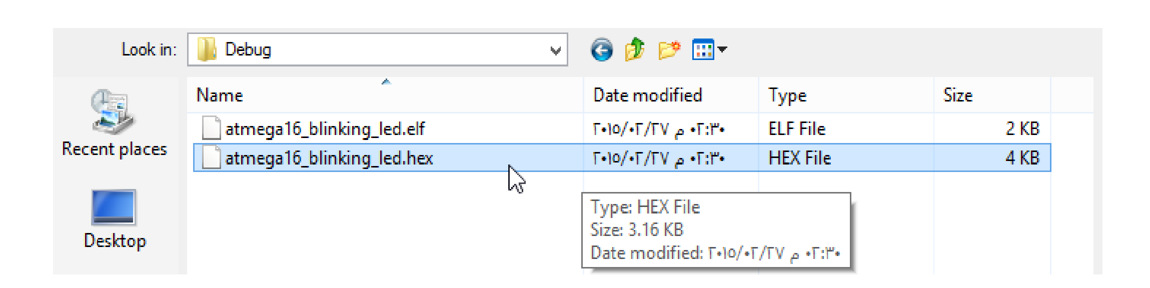




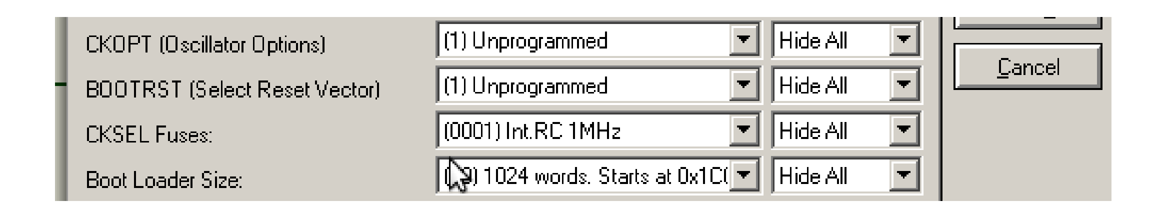
بذلك نكون قد انتهينا من توصيل المكونات الأساسية ويتبقى فقط إضافة ملف الهيكس hex الخاص بالكود الذي كتبناه على برنامج CodeBlocks وذلك عبر الضغط بالزر الأيمن على المُتحكِم ATmega16 واختيار "تعديل خصائص المُتحكِم" ثم الضغط على Program File واختيار ملف الهيكس الذي صنعناه باستخدام Atmel stduio (ستجد الملف في مجلد المشروع) كما في الصورة التالية:

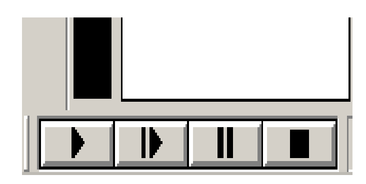




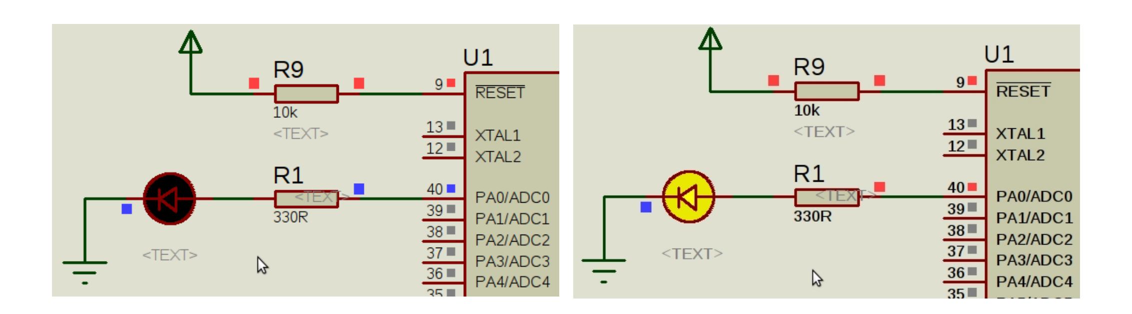


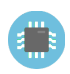
وأخيراً قم بتعديل CKSEL Fuses باختيار Int. RC 1 Mhz كما في الصورة التالية





قم بحفظ الإعدادات عبر الضغط على زر OK ثم قم بتشغيل المحاكاة عبر الضغط على زر play في الشريط السفلي لبرنامج بروتس والآن يفترض أن يضيء الدايود الضوئي لمدة نصف ثانية وينطفئ لمدة نصف ثانية أخرى.





4.2 شرح المثال الأول وأساسيات برمجة الـ AVR

عادة ما تم تنقسم البرامج البسيطة للمتحكمات إلى 3 أجزاء أساسية:

- استدعاء المكتبات وتعريف الثوابت
- الدالة الرئيسية للبرنامج Main Function
- الدوال الإضافية "إن وجدت"

هيكل البرامج

الشكل التالي يوضح الهيكل الرئيسي لمعظم البرامج الخاصة بمتحكمات AVR

استدعاء المكتبات والتعريفات

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <avr/delay.h>
```

البرنامج الرئيسي

```
int main(void) {
   DDRA = 0b00000001;

   while(1) {
           PORTA = 0b00000001;
          _delay_ms(500);
           PORTA = 0b00000000;
          _delay_ms(500);
   }

   return 0;
}
```

في الجزء الأول من البرنامج نجد الأمر #define F_CPU 1000000UL والتي تعني تعريف الثابت **F_CPU** بقيمة =1000000 وهذه السرعة تعبر سرعة المعالج الداخلي (التردد الذي يعمل به المعالج داخل المُتحكِم الدقيق). يجب دائماً أن نضع هذه العبارة في بداية أي برنامج للمتحكمات الدقيقة، وكما سنرى في الفصول المتقدمة أنه يمكننا تغير هذا الرقم وكذلك سرعة المعالج من 1 ميجاهرتز إلى 16 ميجاهرتز.

ملاحظة: الرمز UL في العبارة #define F_CPU 1000000UL يعني كلمة unsigned long وتستخدم للتحكم في حجم الثوابت والمتغيرات كما سنرى في الفصل القادم.

في السطر الثاني والثالث. قمنا باستدعاء مكتبتين وهما avr/**io.h** و util/**delay.h** ويتم ذلك باستخدام الأمر الخاص باستدعاء المكتبات **#include** ثم يكتب اسم المكتبة داخل قوسين <....>

```
#include <avr/io.h>
#include <util/delay.h>
```

- المكتبة الأول **io.h** هي المكتبة المسؤولة عن الـ GPIO والتحكم بها وكذلك تسمية كل مخرج باسم خاص به مثل PORTA و PB0 أو PC1 ... الخ (كما سنرى بالتفصيل في التجارب القادمة).
- المكتبة الثانية **delay.h** هي المسؤولة عن التلاعب بالزمن وحساب الوقت الذي يمر على تشغيل المعالج وهي المكتبة التي تمكننا من إضافة تأخير زمني أو التحكم في وقت تشغيل أي مخرج.

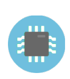
> ملاحظة: كلمتي **avr** و **util** الموجودة قبل أسماء المكتبات تعبر عن أسماء "المجلدات Folders" التي تتواجد بها هذه المكتبات، حيث قامت شركة ATmel بتوزيع المكتبات على مجلدات لتسهيل عملية تصنيفها.

الجزء الثاني من البرنامج هو الدالة Main والتي ستحتوي بداخلها على البرنامج الحقيق الذي يتم تشغيله على المُتحكِم الدقيق. غالبا ما يتم تقسيم الـ Main إلى جزأين كالتالي:

- الإعدادات الخاصة بالمُسجِلات Registers configurations
- البرنامج الذي يتم تشغيله باستمرار while loop

```
int main(void) {
  // هنا تكتب إعدادات المُسجِلات

while(1)
  {
        // هنا تكتب كافة الأكواد البرمجية
        // التي سيتم تنفيذها بصورة مستمرة على المُتحكِم الدقيق
  }
  return 0;
}
```



إعدادات مُسجلات الدخل والخرج الرقمي Digtial I/O

تمتلك مُتحكِمات AVR عدد 3 مسجلات أساسيات للتحكم في أي بورت والتي يتم ضبطها في الـ Main function أشهر هذه المُسجِلات هي:

DDR x → Data Direction Register.
PORT x → Port Output Register.
PIN x → Port Input Register.

المُسجِل DDRx Register

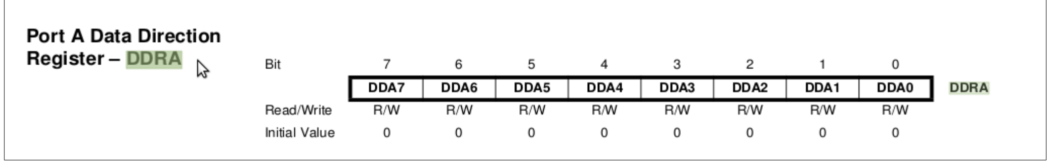
DDRx هو مُسجِل 8 بت يتحكم في **"اتجاه البيانات"** ويعتبر المسؤول عن التحكم في أطراف أي بورت لتعمل إما كدخل Input أو خرج Output، حرف الـ x في نهاية اسم المُسجِل (وكذلك جميع المُسجِلات) يعبر عن أحد الرموز A,B,C,D وهي أسماء البورتات. فمثلا **DDRC** هو مُسجِل اتجاه البيانات للبورت C والمسجل **DDRA** هو الخاص بالبورت A وهكذا ..

كل بت داخل هذا المُسجِل تتحكم في أحد الأطراف الخاصة بالبورت حيث يعبر رقم **1** عن أن هذا الطرف يعمل كخرج output أما **0** فيعبر عن أن هذا الطرف يعمل كدخل input.

عندما نضبط أحد الأطراف لتعمل كخرج فهذا يعني أنه يمكن توصيل أي عنصر إلكتروني بهذا الطرف والتحكم به خلال إرسال إشارات كهربية إلى "output signal" هذه العناصر قد تكون Led, Motor, LCD, Relay, speaker... الخ. كما سنرى في التجارب القادمة.

أما إذا جعلنا هذا الطرف يعمل كدخل عندها يمكنك استقبال إشارة كهربية من خلاله "input signal" مثل الإشارات القادمة من المفاتيح switch أو الحساسات sensors.

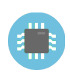
وكما نرى في الجدول التالي (الخاص بالمسجل DDRA صفحة 66 من دليل بيانات ATmega16). نجد أنه يتكون من 8 بت بدءاً من البت رقم 0 إلى البت رقم 7


Port A Data Direction Register – DDRA

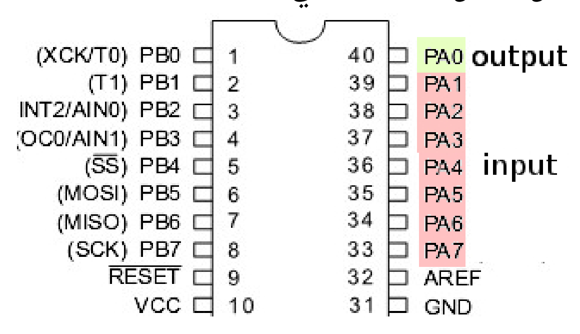
Bit	7	6	5	4	3	2	1	0	
	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	DDRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

يتم التحكم في هذه البتات عبر وضع القيمة المطلوب بها مباشرة مثل أن نكتب الأمر

```
DDRA = 0b00000001;
```



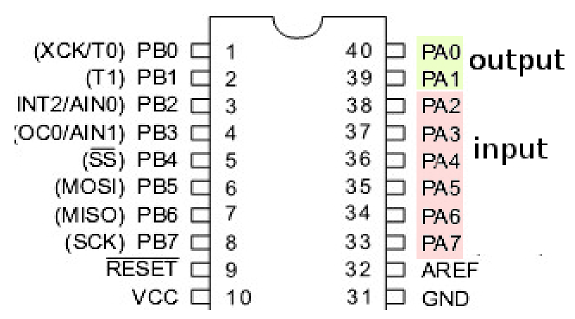
هذا يعني أن نضع القيمة 00000001 داخل المُسجِل DDRA والتي تعني أن البت الأولى فقط = واحد أما باقي البتات = صفر مما يعني أن الطرف PA0 يعني كخرج output أما باقي الأطراف في البورت تعمل كدخل input كما في الصورة التالية:



وإذا قمنا بتعديل الأمر ليصبح:

```
DDRA = 0b00000011;
```

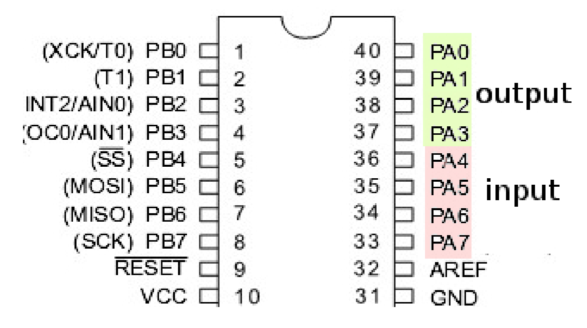
فهذا يعني أن الطرف **PA0** و **PA1** تعمل كخرج أما الأطراف من P2 إلى PA7 تساوي صفر وتعمل كدخل.

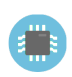


وإذا قمنا بكتابة الأمر السابق ليصبح

```
DDRA = 0b00001111;
```

فهذا يعني أن أول 4 أطراف من البورت PA0,1,2,3 تعمل كخرج وأخر 4 أطراف تعمل كدخل





ملاحظة: عندما نكتب رقم يبدأ ب 0b في لغة السي مثل **0b00011100** فهذا يعني أننا نكتب رقم بالصيغة الثنائية binary أما عندما نكتب رقم يبدأ ب 0x مثل 0xff فهذا يعني أن الرقم مكتوب بالصيغة hexadecimal. ويمكنك أن تتعرف أكثر على أنواع الصيغ عبر قراءة الملحق المسمى **"أساسيات الأنظمة الرقمية".**

أيضاً لاحظ أن جميع المُسجِلات والأرقام الثنائية تبدأ العد **من اليمين إلى اليسار** وهذا يعني أن ال LSB (أول بت) هي البت الموجود على الطرف الأيمن من الرقم 0bxxxxxxx**X** أما ال MSB (أخر بت) فهي الموجود على الطرف الأيسر بعد الحرف b مباشرة 0b**X**xxxxxxx

كما هو ملاحظ في الصورة الخاصة بالمسجل DDRA سنجد هناك كلمة تسمى Initial value والتي تعني القيم الافتراضية لكل البتات والتي تساوي صفر مما يعني أن جميع الأطراف تعمل بصورة افتراضية كدخل.

أيضاً سنجد أن أسفل كل بت كلمة Read/Write والتي تعني أنه يمكنك تعديل محتوى هذا المُسجِل write كما فعلنا في الأمر DDRA=0b00000001 أو يمكنك قراءة محتواه Read وستتضح هذه الخاصية بالتفصيل في الفصل القادم حيث سنقوم بقراءة هذه المُسجِلات.

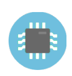
المُسجِل PORTx Register

يتحكم المُسجِل PORTx في الخرج الرقمي لأي طرف، فمثلاً عندما قمنا بتوصيل الدايود الضوئي على الطرف PA0 قمنا بتشغيله وإطفاءه باستخدام هذا المُسجِل، ومثل ال DDRx فإنه يمتلك 8 بت كل بت منهم تتحكم في أحد الأطراف لكل بورت.

الصورة التالية مثال على المُسجِل PORTA (صفحة 66 من دليل البيانات).

Port A Data Register – PORTA

Bit	7	6	5	4	3	2	1	0	
	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	PORTA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	



كل بت في هذا المُسجِل تحمل إما القيمة 0 - LOW أو القيمة 1 – HIGH وعندما يتم وضع القيمة = 1 فهذا يعني أن المُتحكِم **سيخرج إشارة كهربية logic HIGH** والتي ستكون 5 فولت (أو نفس قيمة فارق الجهد الذي يعمل به المُتحكِم). أما إذا كانت القيمة = صفر فهذا يعني أن الطرف سيكون LOW أو 0 فولت وسيتنقل إلى وضع الـ sink mode (سيتم شرح sink mode بالتفصيل في الفصل التالي).

في المثال السابق استخدمنا مجموعة الأوامر

```
while(1)
{
        PORTA = 0b00000001;
        _delay_ms(1000);

        PORTA = 0b00000000;
        _delay_ms(1000);
}
```

هذه الأوامر كانت تستخدم للتلاعب بالقيم الخاصة بالمسجل PORTA كالتالي:

الأمر PORTA = 0b00000001 يعني تغيير قيمة البت الخاصة بالطرف PA0 لتساوي **1** أما باقي البتات تساوي **0** وهذا يعني إخراج إشارة كهربية بقيمة (5 volt) HIGH على الطرف PA0 والتي ستجعل الدايود الضوئي المتصل بهذا الطرف يضيء نتيجة الإشارة الكهربية أما باقي الأطراف تكون LOW (0 volt)

الأمر _delay_ms(1000) يعني أن المُتحكِم الدقيق سينتظر 1000 مللي ثانية قبل تنفيذ الأمر التالي (لاحظ أن الـ 1000 مللي ثانية = 1 ثانية).

الأمر PORTA = 0b00000000 مثل الأمر السابق ولكن باختلاف أن جميع البتات الآن أصبحت تساوي صفر بما في ذلك البت الخاصة بالطرف PA0 مما سيجعل هذا الطرف يساوي LOW (0 volt) وسيؤدي ذلك إلى انطفاء الدايود الضوئي.

ثم يأتي الأمر _delay_ms(1000) ليجعل المُتحكِم الدقيق ينتظر 1000 مللي ثانية مرة أخرى قبل أن يعاد تنفيذ جميع الأوامر السابقة بسبب الدالة while (1)

نظرة عامة على المثال الأول

الكود التالي هو نفس المثال بعد إضافة تعليقات على كل سطر تشرح وظيفته.

```
#define F_CPU 1000000UL          // تحديد سرعة المعالج
#include <avr/io.h>              // استدعاء المكتبات البرمجية
#include <avr/delay.h>

int main(void)
{
   DDRA = 0b00000001;            // تفعيل الطرف الأول ليعمل كخرج

while(1)                         //  استمر في هذا البرنامج إلى ما لا نهاية
{
        PORTA = 0b00000001;      //قم بتشغيل البت الأولى
        _delay_ms(1000);         // انتظر 1000 مللي ثانية
        PORTA = 0b00000000;      // قم بإطفاء البت الأولى
        _delay_ms(1000);         //انتظر 1000 مللي ثاني
}

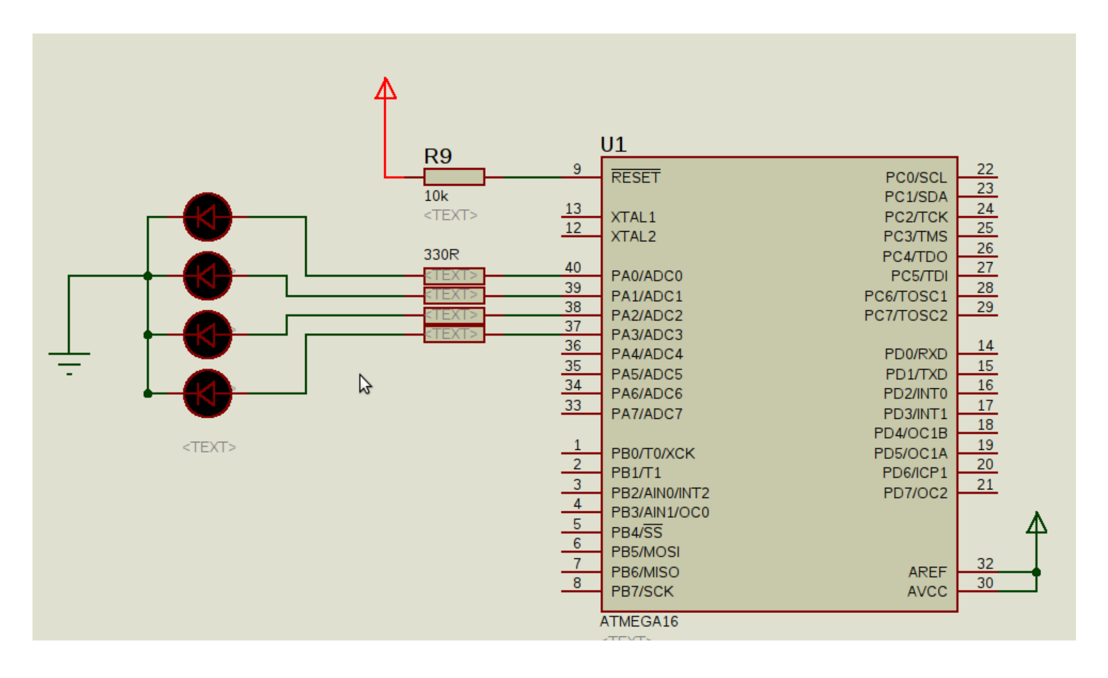
return 0 ;
}                                // نهاية البرنامج
```

ملاحظة: العلامات // أو العلامات /* */ تعني أن الكلام المكتوب هو تعليق comment ولا يحتسب ضمن أكواد البرنامج، ويعتبر استخدام التعليقات أمر هام جداً لتوضيح الأكواد. لذا أنصحك أن تكتب دائماً تعليق على كل سطر برمجي أو دالة في برنامجك.

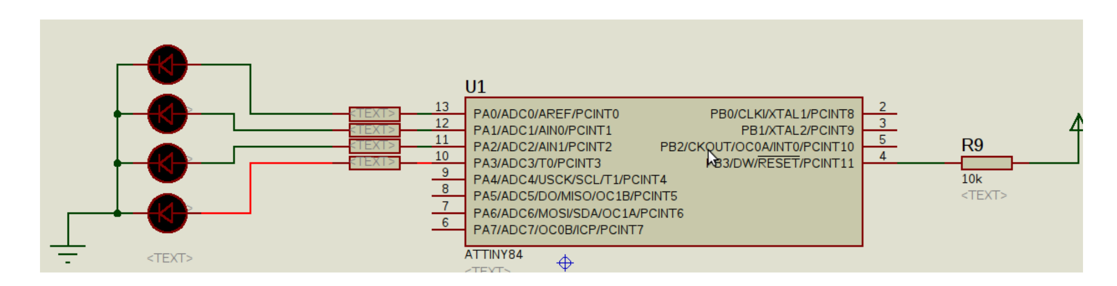
العديد من محترفي البرمجة قد يقومون بكتابة التعليقات حتى قبل البدء في كتابة الأكواد نفسها ويساعدهم ذلك على تنظيم الأفكار وتحديد ما يجب أن يكتب بصورة منظمة، لذا احرص دائماً على توضيح وشرح كل سطر برمجي تكتبة باستخدام comment يسبق هذا السطر

4.3 المثال الثاني: استخدام 4 دايود ضوئي

في هذا المثال سنستخدم 4 دايوادت ضوئية حيث سنقوم بتطوير الكود المستخدم في المثال الأول ليعمل بعدد **4 من الدايودات ضوئية**. وكما هو موضح في الصورة التالية نجد الدايودات متصلة على الأطراف من PA0 إلى PA3 سواء كنت تستخدم ATmega16 أو ATtiny84.



في حالة استخدام ATTiny84 تكون التوصيلات كالتالي:



الكود البرمجي

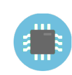
```
#define F_CPU 1000000UL                          // تحديد سرعة المعالج
#include <avr/io.h>
#include <avr/delay.h>                           // استدعاء المكتبات البرمجية
int main(void)
{

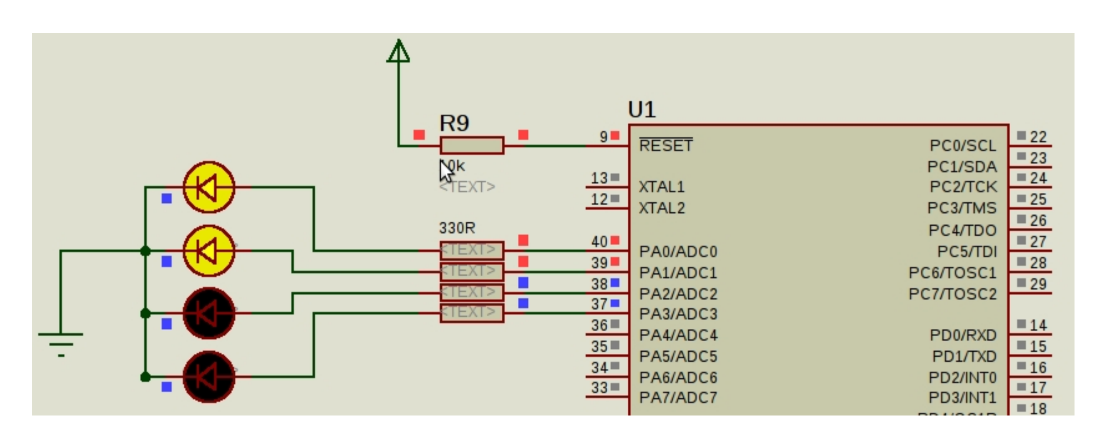
   DDRA = 0b00001111;                            // تفعيل أول 4 أطراف كمخرج

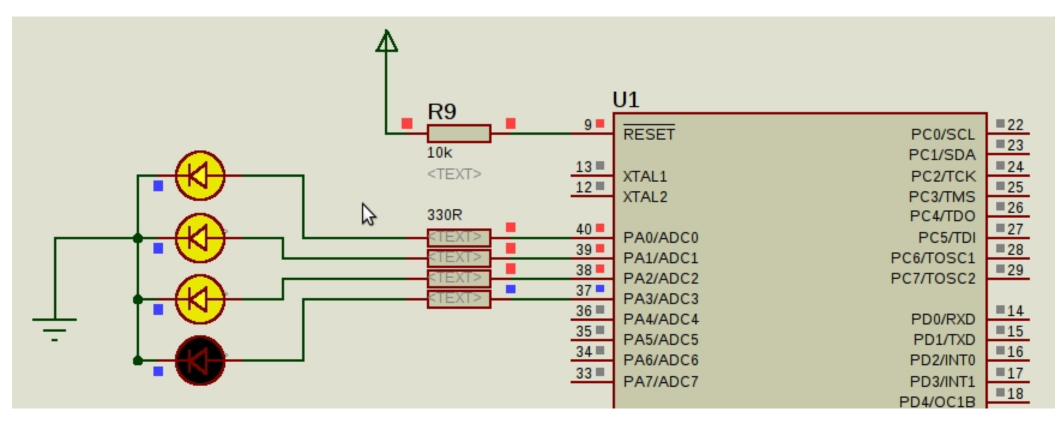
   while(1)
   {
      PORTA = 0b00000001;                        // شغل الطرف الأول
      _delay_ms(500);                            // انتظر نصف ثانية
      PORTA = 0b00000011;                        // شغل الطرف الثاني مع الأول
      _delay_ms(500);                            // انتظر نصف ثانية
      PORTA = 0b00000111;                        // شغل الطرف الأول، الثاني والثالث
      _delay_ms(500);                            // انتظر نصف ثانية
      PORTA = 0b00001111;                        // شغل أول أربعة أطراف من البورت
      _delay_ms(500);                            // انتظر نصف ثانية
   }
   return 0;
}
```

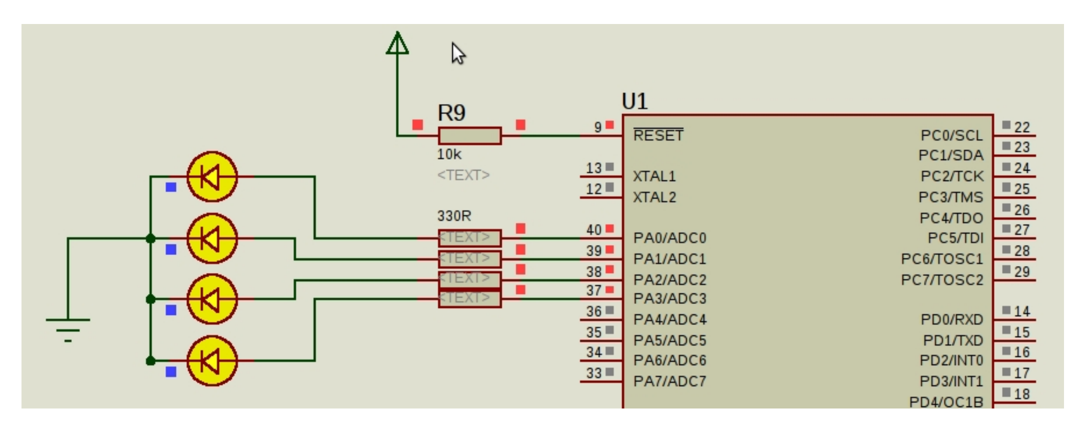
شرح الكود

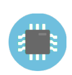
كما نرى في المثال بالأعلى، يعتبر مطابق لنفس المثال الأول باختلاف أننا استخدمنا 4 دايودات وبالتالي قمنا بضبط المُسجِل DDRA ليجعل أول 4 اطراف تعمل كخرج PA0, PA1, PA2, PA3 وبالتالي يمكن استغلالها في تشغيل الدايوادات الأربعة. ثم يأتي الكود المكتوب داخل while (1) والذي يقوم بتغير محتوى PORTA بصورة تصاعدية بحيث يشغل دايود واحد كل 500 مللي ثانية (نصف ثانية). والصور التالية توضح ما سيحدث للدايودات.




R9
10k
<TEXT>
330R
U1
RESET
XTAL1
XTAL2
PA0/ADC0
PA1/ADC1
PA2/ADC2
PA3/ADC3
PA4/ADC4
PA5/ADC5
PA6/ADC6
PA7/ADC7
PC0/SCL
PC1/SDA
PC2/TCK
PC3/TMS
PC4/TDO
PC5/TDI
PC6/TOSC1
PC7/TOSC2
PD0/RXD
PD1/TXD
PD2/INT0
PD3/INT1

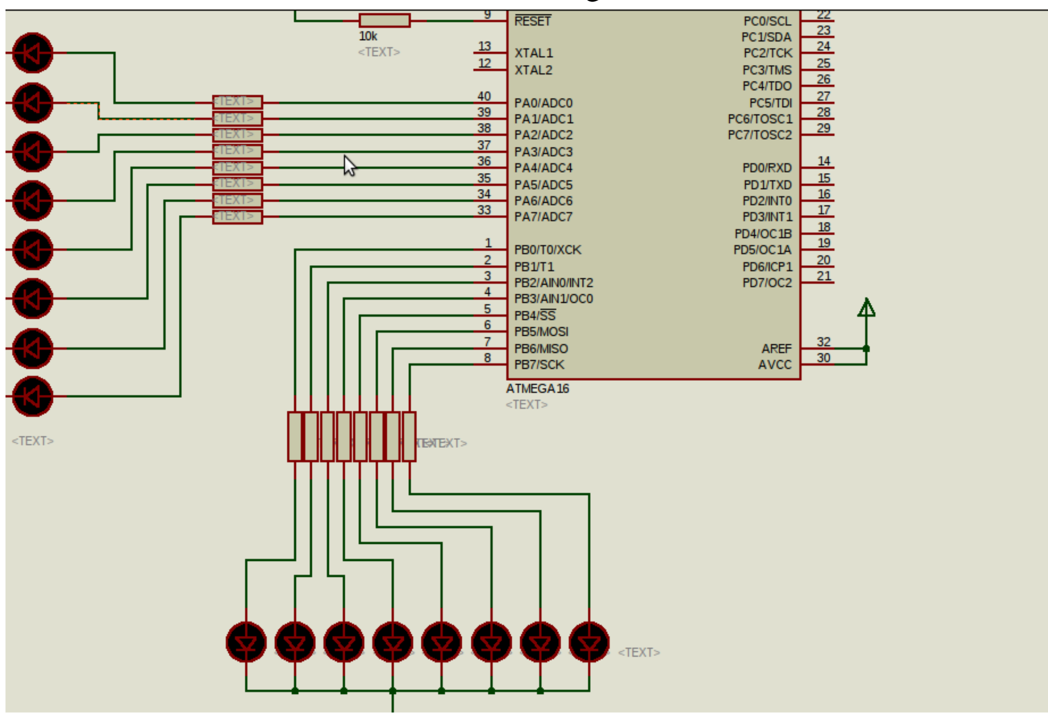

R9
10k
<TEXT>
330R
U1
RESET
XTAL1
XTAL2
PA0/ADC0
PA1/ADC1
PA2/ADC2
PA3/ADC3
PA4/ADC4
PA5/ADC5
PA6/ADC6
PA7/ADC7
PC0/SCL
PC1/SDA
PC2/TCK
PC3/TMS
PC4/TDO
PC5/TDI
PC6/TOSC1
PC7/TOSC2
PD0/RXD
PD1/TXD
PD2/INT0
PD3/INT1
PD4/OC1B


R9
10k
<TEXT>
330R
U1
RESET
XTAL1
XTAL2
PA0/ADC0
PA1/ADC1
PA2/ADC2
PA3/ADC3
PA4/ADC4
PA5/ADC5
PA6/ADC6
PA7/ADC7
PC0/SCL
PC1/SDA
PC2/TCK
PC3/TMS
PC4/TDO
PC5/TDI
PC6/TOSC1
PC7/TOSC2
PD0/RXD
PD1/TXD
PD2/INT0
PD3/INT1
PD4/OC1B

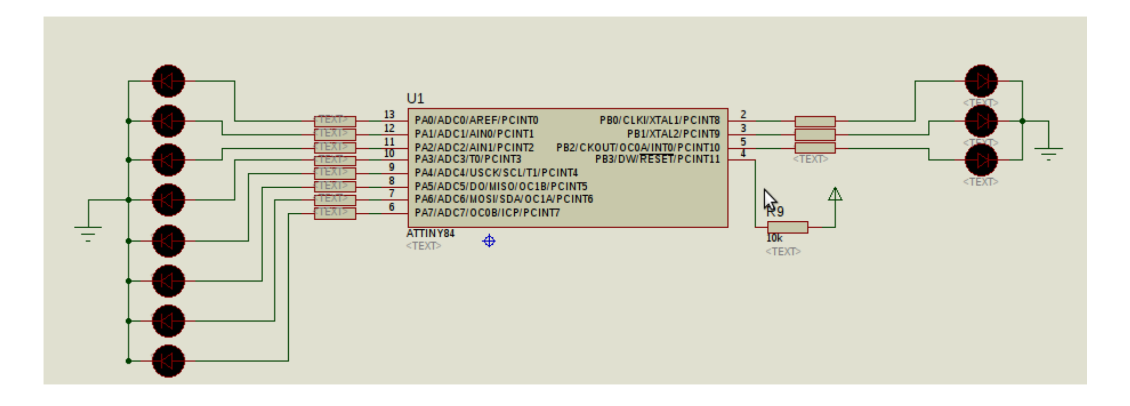


4.4 المثال الثالث: تشغيل جميع أطراف PortA, Port B

في هذا المثال سنقوم بتوصيل 16 دايود على جميع أطراف البورت A و B بحيث يتصل 8 دايودات ضوئية لكل بورت كما هو موضح بالصورة التالية



في حالة المُتحكِم ATTiny84 سيتم توصيل 11 دايود فقط (3+8) لأن البورت A يمتلك 8 أطراف بينما البورت **B يمتلك 3 أطراف فقط** كما هو موضح في الصورة التالية



الكود البرمجي

```
#define F_CPU 1000000UL          // تحديد سرعة المعالج
#include <avr/io.h>
#include <avr/delay.h>           // استدعاء المكتبات البرمجية

int main(void)
{
  DDRA = 0b11111111;             // تفعيل جميع أطراف Port A كخرج
  DDRB = 0b11111111;             // تفعيل جميع أطراف Port B كخرج

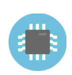
  while(1)
  {
    PORTA = 0b11111111;   // شغل جميع أطراف Port A
    PORTB = 0b11111111;   // شغل جميع أطراف Port B
    _delay_ms(500);       // انتظر نصف ثانية

    PORTA = 0b00000000;   // إطفاء جميع أطراف Port A
    PORTB = 0b00000000;   // إطفاء جميع أطراف Port A
    _delay_ms(500);       // انتظر نصف ثانية
  }
  return 0;
}
```

شرح الكود

في البرامج السابق قمنا باستخدام المسجلين DDRA و DDRB لتشغيل جميع أطراف البورت A والبورت B لتعمل كخرج. ثم قمنا باستخدام المُسجلين PORTA و PORTB لتشغيل جميع الـ Leds على هذه الأطراف لمدة نصف ثانية ثم إطفائها لنصف ثانية. وهكذا إلى ما لا نهاية.

الصورة التالية تمثل محاكاة البرنامج السابق على بروتس مع المُتحكِم ATmega16

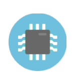


تمارين إضافية

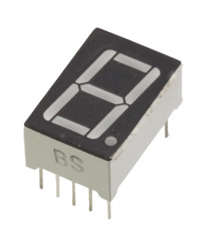
- قم بتشغل 8 دايودات ضوئية على جميع أطراف البورت A واجعلها تضيء بالترتيب التالي مع تأخير ربع ثانية فقط بين كل أمر. (لا تنسى أن جميع أطراف البورت ستعمل كخرج)

00000001
00000011
00000111
00001111
10000000
11000000
11100000
11110000

- ماهو أقصى عدد من الدايودات الضوئية يمكن توصيله بالمتحكم ATmega16 والمتحكم ATtiny84 ؟



4.5 المثال الرابع: تشغيل المقاطعة السباعية 7segment



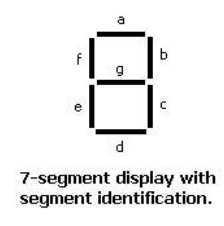
"المقاطعة السباعية - تنطق سيفين سيجمنت" **7segment** وهي عبارة عن مستطيل صغير يحتوي على 7 مقاطع مضيئة باستخدام دايودات ضوئية (متوفرة باللون الأحمر والأخضر والأزرق).وتستخدم في عرض الأرقام وبعض حروف اللغة الإنجليزية.

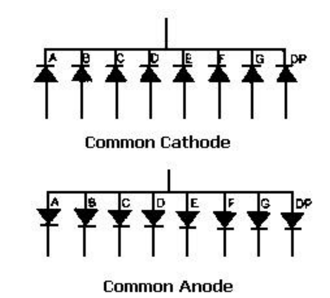
تتوفر هذه القطعة الإلكترونية في الأسواق بمختلف الأحجام فمنها ماهو صغير جداً مثل المستخدمة في الساعات الرقمية الرخيصة ومنها ماهو كبير الحجم مثل المستخدمة في إشارات المرور (اللوحة المضيئة التي تعرض الوقت المتبقي لتفتح إشارة المرور).





تتكون السيفين سيجمنت من 7 دايودات ضوئية متصلة ببعضها البعض إما عن طريق توصيل الطرف الموجب وتسمى common anode أو عن طريق توصيل الطرف السالب وتسمى common cathode **(سنستخدم في التجارب التالية النوع common cathode)**.ويسمى كل دايود ضوئي بأحد حروف الأبجدية الإنجليزية A,B,C,D,E,F,G كما هو موضح في الصورة التالية:





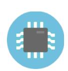
لتشغيل هذا العنصر الإلكتروني سنقوم بتوصيل الأطراف السبعة a,b,c,d,e,f,g بأحد البورتات في المُتحكِم الدقيق (سنستخدم البورت A).

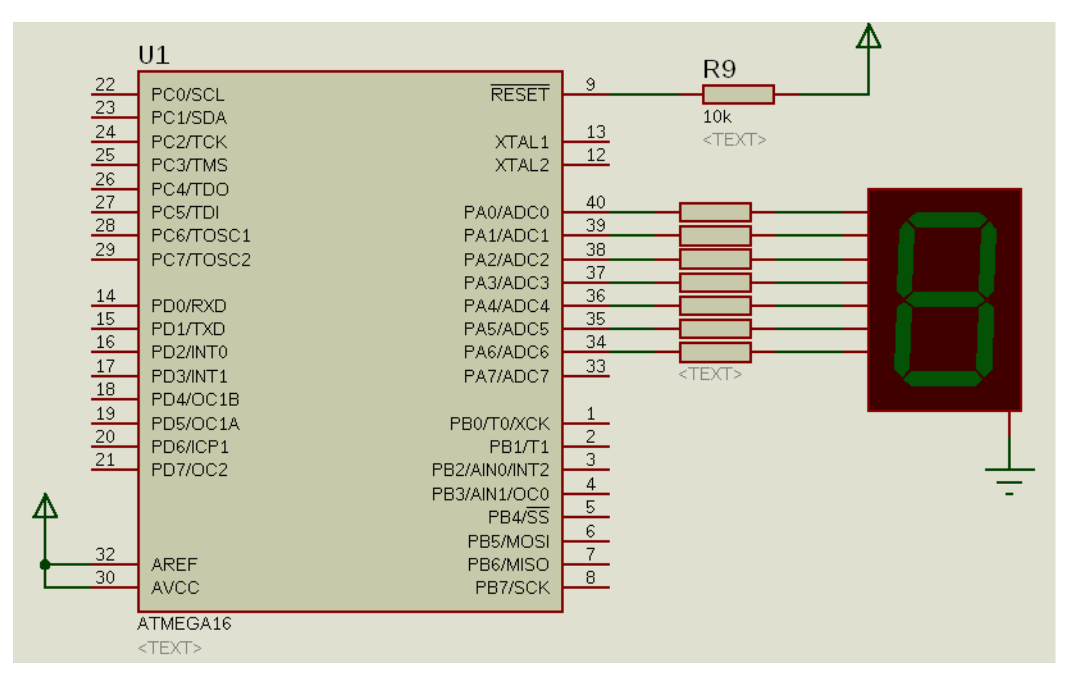
ملاحظة: بعض السيفن-سيجمنت المتوفرة في الأسواق **(خاصة صغيرة الحجم)** تحتوي على طرف إضافي وهو دايود ضوئي صغير موجود على الجانب الأيمن السفلي ويستخدم في بعرض الفاصلة العشرية (.) لكن برنامج بروتس لا يحتوي على هذا الدايود الضوئي لذا لن نستخدمه في المحاكاة).

لعرض أي رقم من الأرقام العشرية سنستخدم الجدول التالي والذي يوضح الحالة التي يجب أن يكون عليها كل دايود ضوئي حتى يتم عرض رقم معين.

DIGIT	LEDs TO GLOW						
	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

الشكل التالي يوضح طريقة توصيل المُتحكِم ATmega16 بالسيفين-سيجمنت على البورت A. وسنقوم بكتابة كود بسيط يعرض الأرقام من 0 إلى 9 بالترتيب وبتأخير زمني 1 ثانية بين كل رقم.





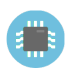
الكود البرمجي

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
   DDRA =0b11111111;

   while(1)

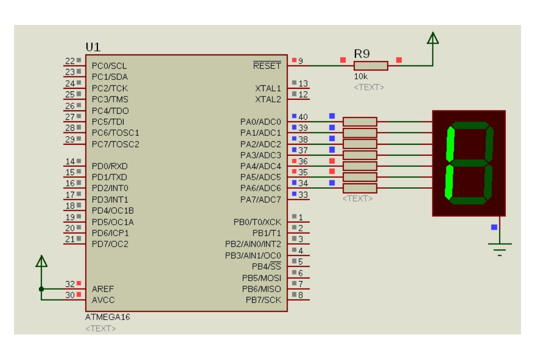
   {
         PORTA = 0b00111111;        // Number 0
         _delay_ms(1000);
         PORTA = 0b00110000;        // Number 1
         _delay_ms(1000);
         PORTA = 0b01011011;        // Number 2
         _delay_ms(1000);
         PORTA = 0b01001111;        // Number 3
         _delay_ms(1000);
         PORTA = 0b01100110;        // Number 4
         _delay_ms(1000);
```

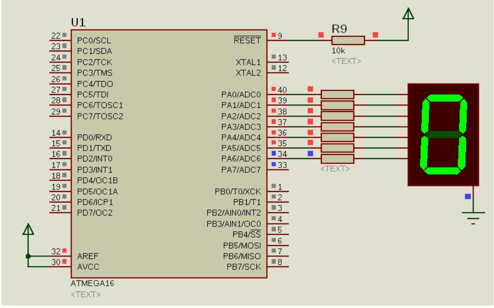


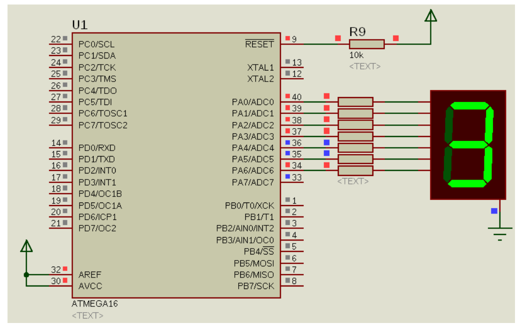
```
        PORTA = 0b01101101;       // Number 5
        _delay_ms(1000);
         PORTA = 0b01111101;      // Number 6
        _delay_ms(1000);
        PORTA = 0b00000111;       // Number 7
        _delay_ms(1000);
        PORTA = 0b11111111;       // Number 8
        _delay_ms(1000);
        PORTA = 0b01101111;       // Number 9
        _delay_ms(1000);

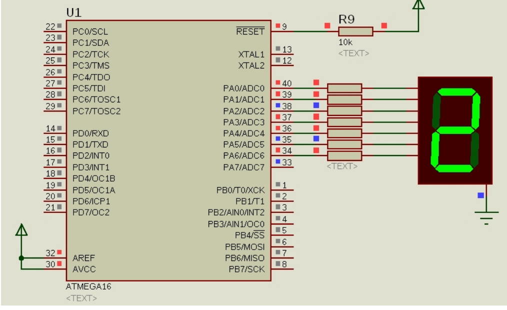
   }
   return 0;
}
```

الصور التالية تمثل المحاكاة بعد ترجمة الكود السابق.









شرح الكود

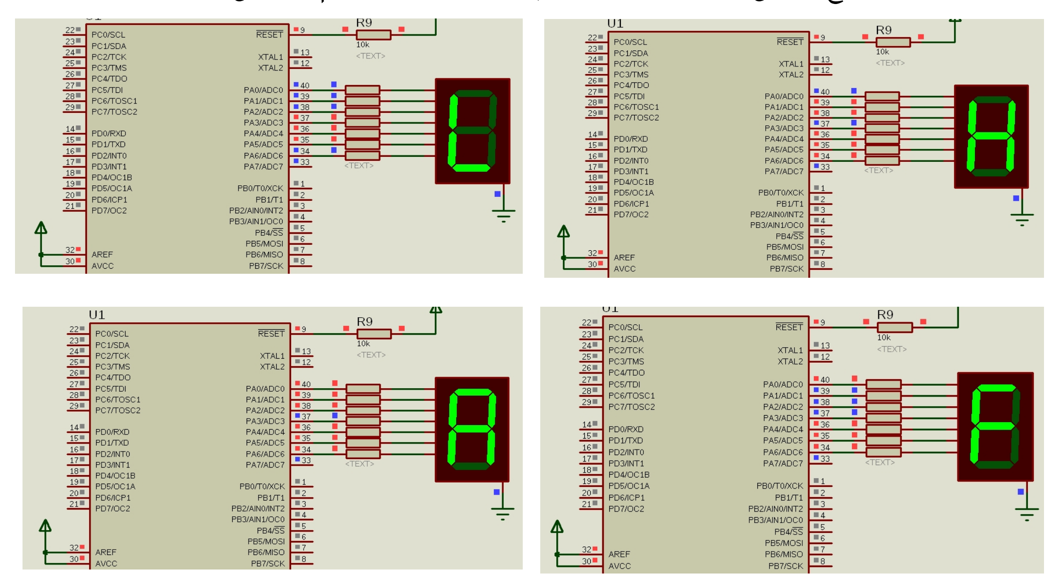
البرنامج السابق قام بتشغيل السيفن سيجمنت بحيث تعرض جميع الأرقام من 0 إلى 9 بصورة متتابعة وذلك عبر كتابة قيمة الرقم المطلوب داخل المُسجِل PORTA.

هناك ملاحظة هامة حول هذا الكود وهي أن الأرقام التي يتم وضعها داخل المُسجِل PORTA تعتبر معكوسة عن الجدول المكتوب بالأعلى وذلك بسبب أن الأطراف التي قمنا بتوصيلها على برنامج بروتس تم عكسها فبدلاً من توصيل a,b,c,d,e,f,g تم توصيلها g,f,e,d,c,b,a.

أيضاً يمكنك كتابة بعض الحروف الإنجليزية البسيطة مثل A,C,F,E,H,L كل ما عليك فعله هو إضافة الجزء التالي للكود بالأعلى (داخل الـ while loop).

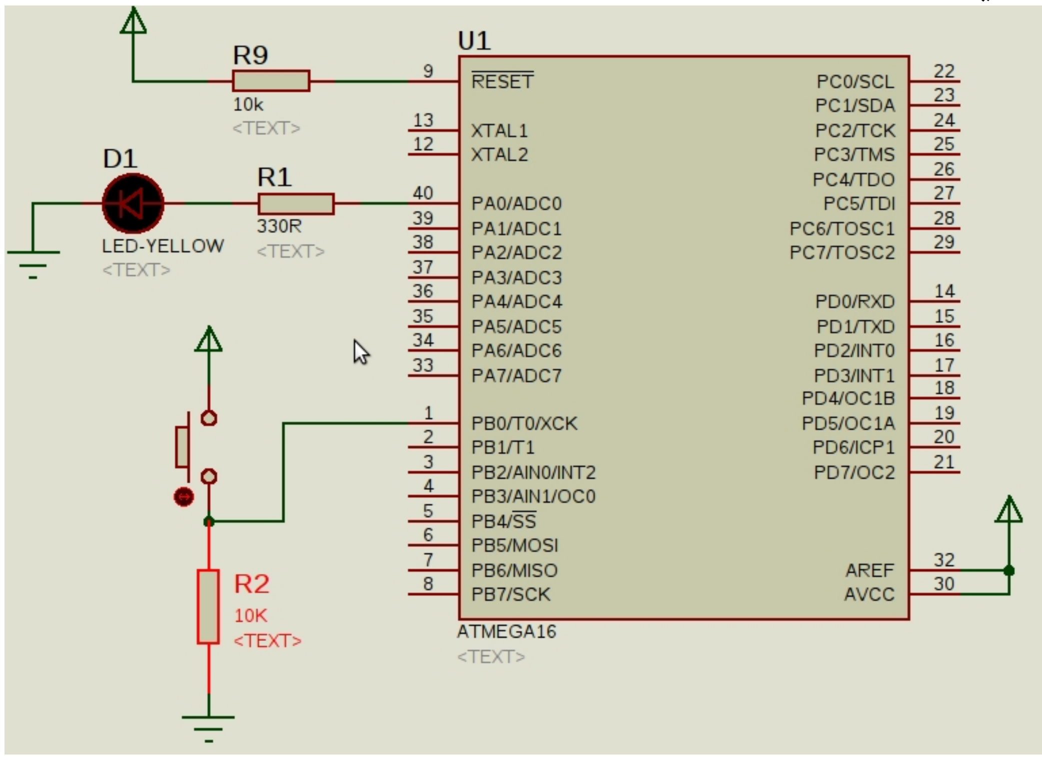
```
PORTA = 0b01110111;        // Letter A
_delay_ms(1000);
PORTA = 0b00111001;        // Letter C
_delay_ms(1000);
PORTA = 0b01110001;        // Letter F
_delay_ms(1000);
PORTA = 0b00111000;        // Letter L
_delay_ms(1000);
PORTA = 0b01110110;        // Letter H
_delay_ms(1000);
```

الصور التالية توضح عرض حروف الأبجدية الإنجليزية باستخدام السيفن-سيجمنت

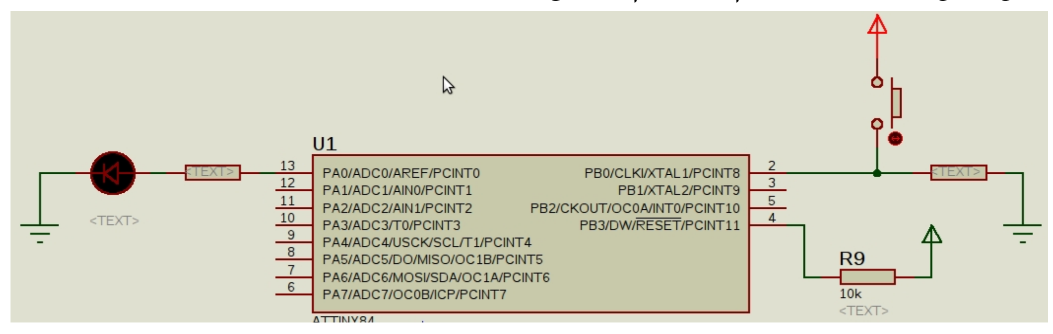


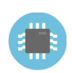
4.6 المثال الخامس: قراءة الدخل الرقمي Inputs reading

في هذا المثال سنتعرف على طرق قراءة الدخل الرقمي Digital Inputs وذلك عبر مجموعة من التجارب باستخدام المفاتيح Switchs. سيتم استخدام مفتاح الضغط Push button مع مقاومة 10 كيلو اوم متصلة على الأرضي كدخل للطرف PB0 وذلك للتحكم بتشغيل دايود ضوئي متصل على الطرف PA0 كما هو موضح بالصورة التالية:



لعمل نفس الدائرة باستخدام المُتحكِم الدقيق ATTiny84





الكود البرمجي

```
#define F_CPU 1000000UL
#include <avr/io.h>

int main(void)
{
   DDRA =0b00000001;
  while(1)
  {
       if (PINB == 0b00000001)
              {PORTA = 0b00000001;}
       else
              {PORTA = 0b00000000;}
  }
  return 0;
}
```

قم بترجمة الكود ثم استخدم ملف الهيكس لمحاكاة المشروع، حيث ستجد أنه عند الضغط على المفتاح سنجد الدايود الضوئي يبدأ بالإضاءة Buttn press وعند ترك المفتاح BUTTON release ينطفئ الدايود الضوئي.

شرح الكود

بصورة افتراضية تعمل جميع أطراف المُتحكِمات الدقيقة كدخل Input port (ليس AVR فحسب وإنما معظم المُتحكِمات الدقيقة من مختلف الشركات) لذا نجد الكود السابق لا يقوم بضبط أطراف البورت B لتعمل كدخل، ومع ذلك يمكنك كتابة الأمر DDRB = 0b00000000 للتأكيد أن أطراف المُتحكِم تعمل كدخل.

يقوم المُتحكِم الدقيق بصورة تلقائية بقراءة محتويات جميع الأطراف ويخزنها في المُسجِل PIN x (حيث تمثل x اسم البورت مثل A,B,C,D ... إلخ) فمثلاً المُسجِل PINB يقوم بتسجيل قيمة الجهد الداخل على جميع أطراف البورت B بصورة مستمرة ويتم تحديث هذا المُسجِل بنفس سرعة عمل المُتحكِم الدقيق فمثلاً إذا كان المُتحكِم يعمل بسرعة 1 ميجاهرتز فهذا

يعني أن المُسجِل PINB يتم تحديثة مليون مرة في الثانية وفي كل مرة يقوم بقراءة جميع أطراف البورت B.

يحتوي المُسجِل PINx على 8 بتات كل بت تمثل قراءة الجهد على أحد أطراف البورت، فمثلاً المُسجِل PINB نجد أنه يتكون من البتات التالية (جميع جداول PINx متوفرة في دليل البيانات).

Port B Input Pins Address – PINB

Bit	7	6	5	4	3	2	1	0	
	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

البت رقم صفر PINB0 تمثل قراءة الطرف PB0 والبت رقم 1 تمثل قراءة الطرف PB1 والبت رقم 2 تمثل قراءة الطرف PB2 ... إلخ.

> ملاحظة: المُسجِل PINB وجميع المُسجِلات PINx من نوع Read only مما يعني أنه يمكنك أن تقرأ منها فقط ولا يمكنك أن تغير محتواها بنفسك بكتابة أي اكود برمجية حيث يتم تحديث المُسجِلات تلقائياً.

في حالة عدم تطبيق أي جهد على أطراف المُتحكِم تكون قيمة البتات = صفر ويتم تغير هذه القيمة عند تطبيق جهد على الطرف الموازي لكل بت. مثلاً لو قمنا بتطبيق جهد 5 فولت على البت PB0 و PB1 و PB2 سنجد أن قيمة المُسجِل PINB تساوي 0b00000111 وإذا قمنا بتطبيق جهد على جميع البتات سنجد القيمة أصبحت 0b11111111 وهكذا ...

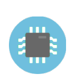
في البرنامج السابق استخدمنا الجملة الشرطة if (condition) لتشغيل الدايود الضوئي عند الضغط على الزر حيث كتبنا الأمر.

```
        if(PINB == 0b00000001)
                {PORTA = 0b00000001;}
```

والذي يعني أنه إذا كانت قيمة المُسجِل PINB تساوي 0b00000001 (يعني تم الضغط على الزر المتصل بـ PB0) قم بتشغيل الدايود الضوئي عبر الأمر PORTA = 0b00000001

```
        else
                {PORTA = 0b00000000;}
```

وفي حالة عدم تطبيق هذا الشرط else قم بإطفاء الدايود عبر PORTA = 0b00000000

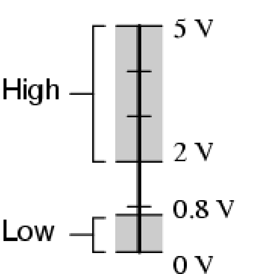


Pull Up & Pull Down Resistor 4.7

من أشهر الكلمات التي قد تسمعها في عالم الإلكترونيات الرقمية هي المقاومة رافعة الجهد أو خافضة الجهد Pull-Up & Pull-Down هذه المقاومات التي تتراوح قيمتها بين 2.2 كيلواوم إلى 10 كيلوأوم تستخدم بصورة أساسية في دخل الأنظمة الرقمية Digital Inputs بداية من البوبات المنطقية البسيطة مثل AND, OR, NOT وانتهائاً بالمُتحكِمات الدقيقة.

إذاً لماذا استخدامنا هذا النوع من المقاومات مثل المقاومة الـ 10 كيلو أوم مع المفتاح في المثال السابق؟ ولماذا لم نوصل المفتاح بالـ VCC مباشرة؟ هناك **3 أسباب** لهذا الأمر..

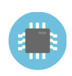
الاستخدام الأول (إلغاء الدخل العائم): جميع المكونات الإلكترونية الرقمية التي تتعامل بالصفر والواحد تعاني من مشكلة خطيرة جداً تسمى المنطقة العائمة Floating Area **أو** Nosie Margin (نطاق الشوشرة) هذه المنطقة هي فارق الجهد بين الـ Logic (0) والـ Logic (1) وتستخدم المقاومات pull-up or down في حل هذه المشكلة.



والسبب في ذلك أن معظم المكونات الإلكترونية تعتبر أي جهد دخل input voltage بين صفر و 0.8 فولت هو 0 أو كما يسمى Low بينما أي جهد بين 2 وحتى 5 فولت يعتبر 1 رقمي HIGH لكن إذا كان هناك دخل بقيمة 0.9 فولت أو 1.25 أو 1.9 تحدث **مشكلة المنطقة العائمة** حيث لا تستطيع الإلكترونيات الرقمية أن تتعرف على هذا الجهد وبالتالي لا يمكنها أن تحدد بدقة هل هو HIGH(1) أم LOW(0) وتبدأ بالتصرف العشوائي (وقد تصاب الدائرة بنوع من الجنون).

بطبيعة الحال جميع الأجهزة الإلكترونية تصدر نوع من الـ Electric noise شوشرة كهربية بما في ذلك جسد الإنسان لذا عند ترك أي طرف input فإنه يتعرض لشوشرة كهربية بفرق جهد صغير نسبياً مما يتسبب في دخول الجهاز بوضع الـ Floating Area.

وجود مقاومة متصلة بطرف الدخل والطرف الأرضي يضمن تماماً أن الطرف = صفر حتى مع وجود الشوشرة ولا يتم تغيير هذا الجهد إلا عند إدخال جهد كبير نسبياً من المفتاح مثل 5 فولت. وتكون قيمة المقاومة بين 2.2 إلى 10 كيلوأوم.



يمكنك أن تجرب بنفسك تأثير المنطقة العامة عبر المثال السابق حيث يتم تركيب الدائرة كما هي على لوحة التجارب breadboard بدون مقاومة 10 كيلو. ثم حاول أن تضغط الزر وشاهد ماذا سيحدث (ستجد أن المُتحكِم الدقيق يبدأ باتخاذ قرارت عشوائية بمجرد أن تقترب يدك من المُتحكِم).

الاستخدام الثاني (عكس الجهد الداخل): في بعض الحالات يكون مطلوب عكس الجهد الداخل من المفتاح، في المثال السابق تم توصيل المقاومة مع المفتاح بأسلوب Pull-Down مما يجعل المفتاح يُدخل جهد 5 فولت للطرف PB0 عند الضغط عليه وعند ترك المفتاح يكون الجهد صفر فولت (أرضي).

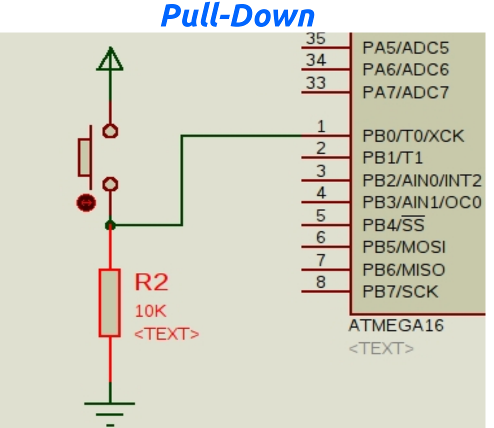
Releas Button = LOW	**Press Button = HIGH**

يمكن استخدام المقاومة الرافعة Pull-up Resistor لعكس هذا الجهد وبذلك يصبح الجهد الأساسي للطرف PB0 هو HIGH وعند ضغط المفتاح يتحول هذا الجهد إلى صفر (أرضي).

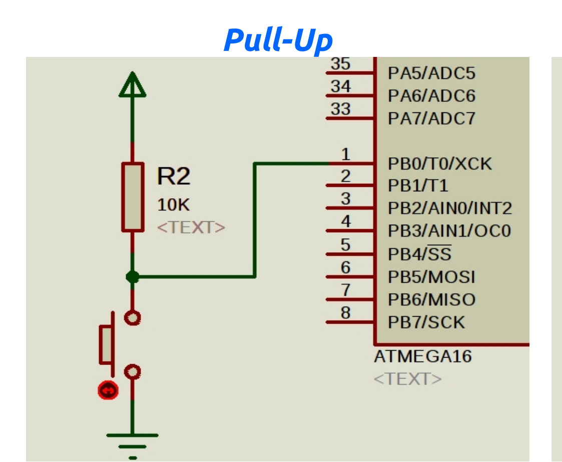
Releas Button = HIGH	**Press Button = LOW**

يتم توصيل المقاومة الرافعة Pull-up بحيث ينعكس مكانها بين الـ VCC والمفتاح والصور التالية توضح الفرق بين توصيله الـ Pull-UP والـ Pull-Down

Pull-Down



Pull-Up



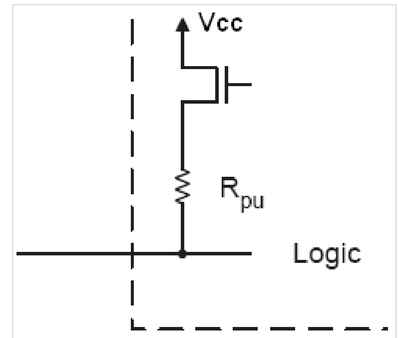
ملاحظة: قد تستخدم مقاومات الـ Pull-Up & Pull-Down مع بروتوكولات الاتصالات مثل i2C لعكس جهد النبضات ويتم استخدام مقاومات بقيمة تتراوح بين 2.2 إلى 4.7 كيلوأوم



الاستخدام الثالث (مُحول تيار إلى جهد): بعض الحساسات الشهيرة مثل الميكروفون Microphone أو المقاومة الضوئية LDR أو الحرارية NTC تقوم بتحويل الحرارة أو الضوء إلى تغير في **التيار** الكهربي وليس تغير في **الجهد** مما يمثل مشكلة في فهم هذه الحساسات. حيث نجد أن جميع المُتحكِمات الدقيقة التي تحتوي على ADC يمكنها قراءة **تغير الجهد فقط** ولا تستطيع التعرف على التيار الكهربي المتغير لذا يتم استخدام مقاومة Pull-Down أو Pull-Up لتحويل التيار المتغير إلى جهد.

4.8 خاصية الـ Internal Pull-Up

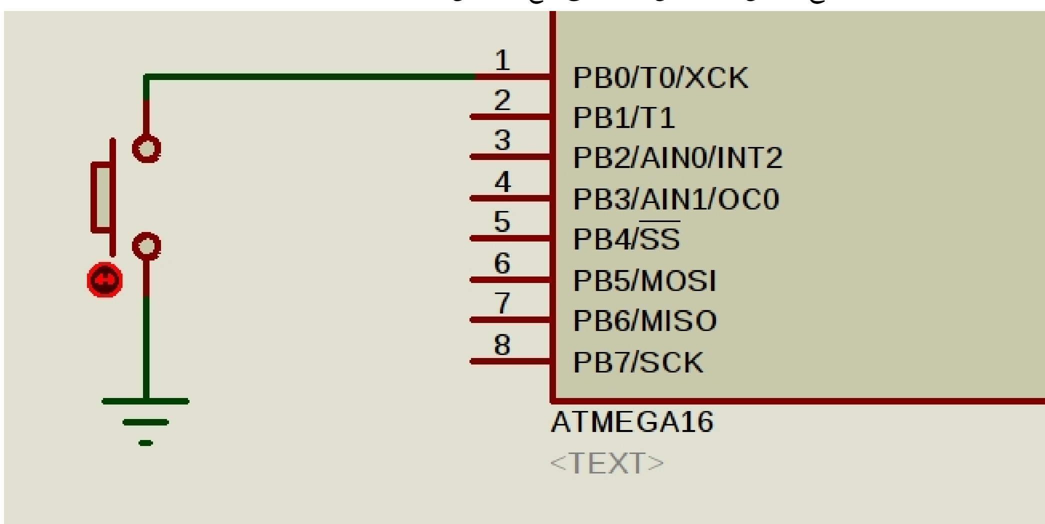
تمتلك معظم مُتحكِمات الـ AVR خاصية جميلة جداً وهي أن أطراف البورتات تمتلك (مقاومة الرفع الداخلية internal pull-up) هذه المقاومة تجعلك تستخدم المفاتيح التي تريدها بدون أي مقاومات إضافية. الشكل التالي يوضح تركيب مقاومة الرفع الداخلية وهي عبارة عن ترانزستور + مقاومة Rpu حيث يتحكم الترانزستور في تفعيل المقاومة أو إلغائها.



بصورة افتراضية يتم إلغاء تفعيل هذه المقاومة ويجب عليك أن تشغلها بنفسك وذلك عبر جعل الطرف المطلوب يعمل كدخل input ثم الكتابة في المُسجِل PORTx لتفعيل هذه المقاومة

لاحظ أن المُسجِل PORTx لن يعمل لكتابة قيم الخرج ولكن هنا سيستخدم <u>لتفعيل الترانزستور</u> **الخاص بمقاومة الـ pull-up فقط.** أيضاً تذكر أنه بعد تفعيل المقاومة يصبح تأثير الضغط على المفتاح معكوس مما يعني أن قراءة PINx تصبح 0 في حالة الضغط على المفتاح وتصبح 1 ترك المفتاح.

الصورة التالية توضح شكل توصيل المفتاح مع تفعيل الـ internal pull-up



4.9 المثال السادس: تشغيل 3 دايودات + 3 مفاتيح

في هذا المثال سنستخدم 3 مفاتيح مع تفعيل خاصية الـ Internal pull-up وبذلك سنوفر استخدام 3 مقاومات 10 كيلو. وسيتحكم كل مفتاح في تشغيل عدد من الدايودات بحيث:

- إذا تم الضغط على المفتاح المتصل بالطرف PB0 يتم تشغيل دايود واحد فقط
- إذا تم الضغط على المفتاح المتصل بالطرف PB1 يتم تشغيل 2 دايود
- إذا تم الضغط على المفتاح المتصل بالطرف PB2 يتم تشغيل 3 دايود

تذكر أنه بعد تشغيل مقاومة الرفع تصبح قراءة المفتاح والمُسجِل **PINx معكوسة** لذا سنجد القيمة الإفتراضية للمسجل PINx = 0b11111111 وعند الضغط على أي مفتاح ستتحول البت المقابلة له إلى صفر، فمثلاً الضغط على المفتاح المتصل ب PB0 سيجعل قيمة PINB تساوي 0b11111110

الكود البرمجي

```
#define F_CPU 1000000UL
#include <avr/io.h>

int main(void)
{
   DDRA =0b00000111;            // تفعيل خرج لتشغيل الثلاث دايودات ضوئية

   DDRB = 0b00000000;           //تأكيد أن جميع أطراف البورت تعمل كدخل
   PORTB = 0b11111111;          // B تفعيل مقاومة الرفع لكل أطراف البورت

  while(1)
  {
        if (PINB == 0b11111110)  {PORTA = 0b00000001;}  // Button 1 pressed
  else if (PINB == 0b11111101)  {PORTA = 0b00000011;}  // Button 2 pressed
  else if (PINB == 0b11111011)  {PORTA = 0b00000111;}  // Button 3 pressed
  else {PORTA = 0b00000000;}
  }

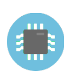
  return 0;
}
```

شرح الكود

في البداية قمنا بتشغيل جميع أطراف البورت B لتعمل كدخل (تذكر أن هذه الخطوة اختيارية وهدفها التأكيد فقط) ثم قمنا بتفعيل جميع مقاومات الـ Pull-UP على أطراف البورت B وذلك عبر الأمر **PORTB** = 0b11111111

ثم بدء البرنامج الأساسي في قراءة المُسجِل PINB ومقارنة محتواه مع مجموعة من الشروط بحيث إذا كانت القيمة = 0b11111110 (تم الضغط على الزر المتصل ب PB0) يتم تشغيل الدايود المتصل بالطرف PA0

وإذا كانت القيمة تساوي 0b11111101 فهذا يعني انه تم الضغط على الزر المتصل بالطرف PB1 وسيؤدي ذلك لتشغيل الدايودات على الأطراف PA0 + PA1 ونفس الأمر مع الزر الثالث (المتصل بالطرف PB2) والذي سيشغل الدايودات الثلاثة PA0, PA1, PA2.



4.10 Bouncing effect & De-bouncing

يعرف تأثير القفز Bouncing effect بأنها ظاهرة تحدث للمفاتيح switchs الميكانيكية خاصة مفاتيح الضغط Push Buttons حيث تتكون هذه المفاتيح عادة من شريحتين من الصفائح المعدنية والتي عند الضغط عليها يحدث بينهم تلامس contact. لكن بسبب طريقة صناعة هذه الصفائح فإنه عند الضغط عليها تقفز الألواح المعدنية أكثر من مرة، هذا الأمر يجعل الصفائح تقوم "بعشرات التلامسات" في الثانية الواحدة قبل أن تستقر.

تعتبر هذه الظاهرة من الأمور المزعجة لمصممي النظم المدمجة وذلك لأنها تجعل المفاتيح الميكانيكية تنتج إشارات غير مطلوبة قد تؤثر على أداء النظام المدمج ككل. من حسن الحظ أنه هناك العديد من الحلول المتوفرة لهذه الظاهرة وتسمى هذه الحلول debouncing

الحل الأول: التأخير الزمني

هذا الحل يعتبر أبسط طريقة وتسمى software debouncing حيث تعتمد هذه الطريقة على التأكد "مرتين" من الضغط على المفتاح مع وضع تأخير زمني بين كل مرة. الكود التالي يمثل تشغيل دايود ضوئي عند الضغط على مفتاح (والثاني)

الكود التقليدي لقراءة المفتاح بدون debouncing

```
if (PINB == 0b11111110) { PORTA = 0b00000001;}
```

قراءة المفتاح باستخدام ال software debouncing (delay)

```
if (PINB == 0b11111110)
{
        delay_ms(10);                //تأخير زمني
        if (PINB == 0b11111110)      // إعادة التأكد أن المفتاح مازال مضغوط
                { PORTA = 0b00000001;}
}
```

هذه الطريقة لها عيوب كثيرة جداً منها أنها لا تحل مشكلة الذبذبات الكهربية نفسها ولكنها تتأكد فقط من ضغط المفتاح لفترة كافية حتى تنتهي تلك الذبذبات كما أنك تضطر إلى إضافة تأخير زمني للبرنامج مما قد يتسبب في تأخير استجابة النظام ككل خاصة إذا كنت

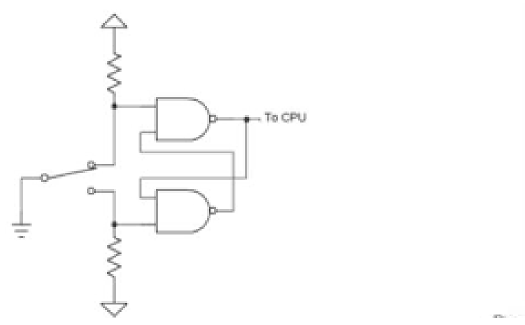
تقرأ أكثر من مفتاح. وفي حالة استخدام نظام تشغيل RTOS لا يمكن استخدام هذا النوع من التأخير الزمني بصورة مباشرة (سيتم شرح التأخير الزمني لنظام ROTS بالتفصيل في الفصل السابع).

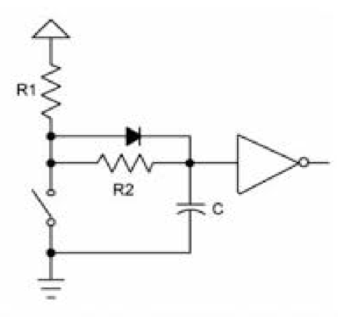
الحل الثاني: استخدام فلتر (مرشح)

هذا الحل يعتمد على استخدام دائرة filter لترشيح الذبذبات الكهربية الناتجة عن الـ bouncing effect وتسمى Debouncing Hardware حيث تستخدم هذه الدائرة في امتصاص الذبذبات الكهربية وتمنع دخولها للمتحكم الدقيق.

أبسط دائرة مرشح هي توصيل مكثف سيراميكي على التوازي مع المفتاح Ceramic - capacitor بقيمة 0.1 ميكروفاراد (يكون مكتوب على المكثف الرقم 104). مع العلم أنه يمكن توصيل هذا المكثف سواء كنت تستخدم المفتاح مع Pull up أو Pull down resistor

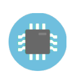
هناك دوائر أخرى أكثر تعقيداً وقوة في الترشيح مثل استخدام مقاومة 10 كيلو + دايود + مكثف (بنفس القيمة). أو باستخدام دوائر رقمية مثل AND gates لعمل digital trigger.





على أي حال، استخدام مكثف واحد فقط كافي لفلترة الذبذبات الكهربية الغير مرغوب فيها ولا داعي لبناء دوائر أكثر تعقيداً كما أنه أوفر من ناحية للتكلفة.

- **المميزات:** الأفضل من ناحية الأداء. متوافقة مع جميع المُتحكِمات الدقيقة ولا تتطلب أي تأخير زمني في الكود كما أنها لن تتسبب في مشاكل للـ Interrupts أو الـ RTOS
- **العيوب:** زيادة التكلفة بسبب استخدام مكونات إضافية مع المفتاح.



4.11 حساب المقاومة المستخدمة قبل الأحمال

الحِمل Load هو أي جهاز يتصل بخرج المُتحكِم الدقيق مثل الدايود الضوئي، في الأمثلة السابقة لاحظنا وجود مقاومة قبل كل دايود بقيمة (330 أوم)، لماذا استخدمنا هذه المقاومة وبتلك القيمة تحديداً؟

هناك سببان لهذا الأمر. الأول هو أن الدايود الضوئي (خاصة اللون الأحمر) يفضل أن يتم تشغيله بتيار كهربي ما بين 15 إلى 20 مللي أمبير عند تطبيق فرق جهد 5 فولت. وبالتعويض في قانون أوم الكهربي (فرق الجهد 5 فولت – التيار 15 مللي أمبير)

$$Voltage(V) = Current(I) \times Resistance(R)$$

نجد أن المقاومة المناسبة هي 330 أوم .

إذا لم تستخدم هذه المقاومة قد تحدث العديد من المشاكل. فمثلاً إذا تعرض الدايود الضوئي الأحمر لأكثر من 25 مللي أمبير لفترة طويلة نسبياً (ساعة أو أكثر) فأنه سيحترق وبالتأكيد أنت لا تريد دائرة إلكترونية تحتاج لتغيير دايود ضوئي كل ساعة !

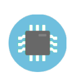
ثانياً: قد يؤدي تشغيل الأحمال بدون مقاومة لفترات طويلة إلى تدمير أطراف (بورتات) المُتحكِم الدقيق. وذلك بسبب محدودية التيار الكهربي الذي يمكن سحبه من كل طرف، إذا نظرت إلى الصفحة الخاصة بالخصائص الكهربية للمتحكم Electrical Charchtaristics (صفحة 291 دليل بيانات المُتحكِم ATmega16). ستجد جدول الـ Absolute values "القيم المطلقة" وهي أقصى قيم كهربية يتحملها المُتحكِم الدقيق.

Electrical Characteristics

Absolute Maximum Ratings*

Operating Temperature	-55°C to +125°C
Storage Temperature	-65°C to +150°C
Voltage on any Pin except RESET with respect to Ground	-0.5V to V_{CC}+0.5V
Voltage on RESET with respect to Ground	-0.5V to +13.0V
Maximum Operating Voltage	6.0V
DC Current per I/O Pin	40.0 mA
DC Current V_{CC} and GND Pins	200.0 mA PDIP and 400.0 mA TQFP/MLF

*NOTICE: Stresses beyond those listed under “Absolute Maximum Ratings” may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or other conditions beyond those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.



لاحظ السطر قبل الأخير حيث نجد DC current per I/O = 40 mA

هذا يعني أن أقصى تيار يمكن سحبة من أي طرف من أطراف المُتحكِم = 40 مللي أمبير وإذا كان الحِمل يسحب أكثر من ذلك فهذا سيؤدي إلى تلف المُتحكِم الدقيق. عليك أن تتذكر هذا الرقم جيداً عند تصميم أي دائرة إلكترونية.

الأفضل أن تسحب الأحمال تيار = 85% من القيمة القصوى وذلك حتى تضمن أن يكون العمر الإفتراضي للمتحكم الدقيق أطول ما يكون. فمثلاً إذا طبقنا هذه القاعدة سنجد أن التيار الآمن هو 0.85 * 40 = 34 مللي أمبير.

<u>**هذا يعني أن أقل مقاومة يمكن توصيلها على أي طرف من أطراف المُتحكِم (من قانون أوم) تساوي 147 أوم، مع ملاحظة أنه يمكن توصيل أي قيمة أعلى لأن المقاومات الأكبر ستمرر تيار كهربي أقل وهذا آمن تماماً.**</u>

توصيل أحمال بتيارات كبيرة

أغلب المُتحكِمات الدقيقة سواء القديمة أو حتى الحديثة نسبياً مثل ARM لا تستطيع أن تشغل أحمال تسحب تيار أعلى من 50 مللي أمبير (أغلب مُتحكِمات ARM يمكنها إمداد الأحمال بتيار ما بين 15 إلى 25 مللي أمبير). لحل هذه المشكلة يتم استخدام دوائر القيادة Driver circuits هذه الدوائر عبارة عن عناصر تعمل كمكبر للجهد أو التيار أو كليهما.

عادة ما تبنى هذه الدوائر باستخدام الترانزستور سواء BJT مثل NPN أو الـ MOSFET ويكون الفارق الأساسي بينهما هو أقصى تيار يمكن أن يتحمله الترانزستور. حيث تتميز ترانزستورات MOSFET بتحمل تيارات قد تصل إلى 40 أمبير على عكس الـ NPN (or PNP) والتي لا تتحمل أكثر من 5 أمبير كأقصى تقدير.

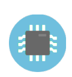
لنأخذ مثال على هذه الدوائر: لنفترض أنك تريد تشغيل 5 دايودات ضوئية ويجب أن تعمل معاً في نفس الوقت، سنجد أنه يتوفر خياران لتشغل هذه الدايودات، الأول هو استخدام 5 أطراف من المتحكم الدقيق وتوصيل كل دايود على طرف مستقل ولكن هذا الخيار يعد إهدار لأطراف المتحكم.

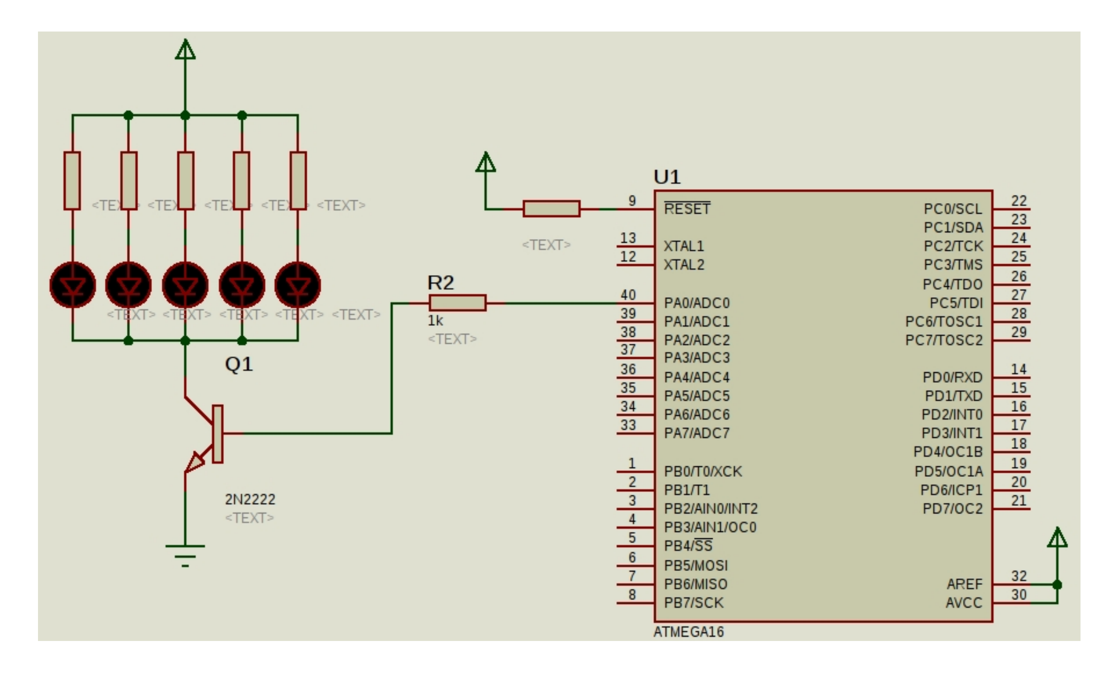
الخيار الثاني هو توصيل الـ 5 دايودات كلها على طرف واحد باستخدام دائرة قيادة (وذلك لأن الدايودات الخمسة ستستهلك 5×15 = 75 مللي أمبير، وهذا أكبر بكثير مما قد يتحمله المتحكم على طرف واحد).

دائرة الترانزستور 2n2222

يُعد هذا الترانزستور أشهر أفراد عائلة الـ NPN وأكثرها توافراً في الأسواق، كما يتميز بالسعر المنخفض (يمكنك شراء نحو 10 قطع منه بدولار واحد) ويستطيع تشغيل أحمال بتيار يصل إلى نصف أمبير (500 مللي)، بافتراض أنك لم تجده في السوق المحلي يمكنك شراء أحد البدائل مثل:

- 2n3904 – مماثل لـ 2n2222 باستثناء أنه يتحمل 380 مللي أمبير فقط
- BC547 - مماثل لـ 2n2222 باستثناء أنه يتحمل 100 مللي أمبير فقط





الدائرة بالأعلى تمثل طريقة توصيل المتحكم الدقيق مع الترانزستور، حيث يتم توصل طرف المتحكم بقاعدة الترانزستور Base من خلال مقاومة يجب أن تكون قيمتها ما بين 150 اوم إلى 1 كيلو اوم.

يتم توصيل الدايودات الضوئية على طرف المجمع collector مع وضع مقاومة 300 اوم قبل كل دايود (لحماية الدايود) ويتم توصيل طرف المشع Emitter بالطرف الأرضي، نظرياً تستطيع هذه الدائرة أن تشغل حتى 30 دايود ضوئي (من النوع أحمر اللون) لكن كما أشرنا مسبقاً يستحسن أن يتم تشغيل أي عنصر إلكتروني بحد أقصى 85% من قدرته الكاملة وهذا يعني أن أقصى تيار للأحمال المتصلة على 2n2222 يجب أن يكون 400 مللي أمبير فقط.

بعد الانتهاء من توصيل هذه الدائرة يمكنك اختبارها عبر نفس المثال الأول Blinking led

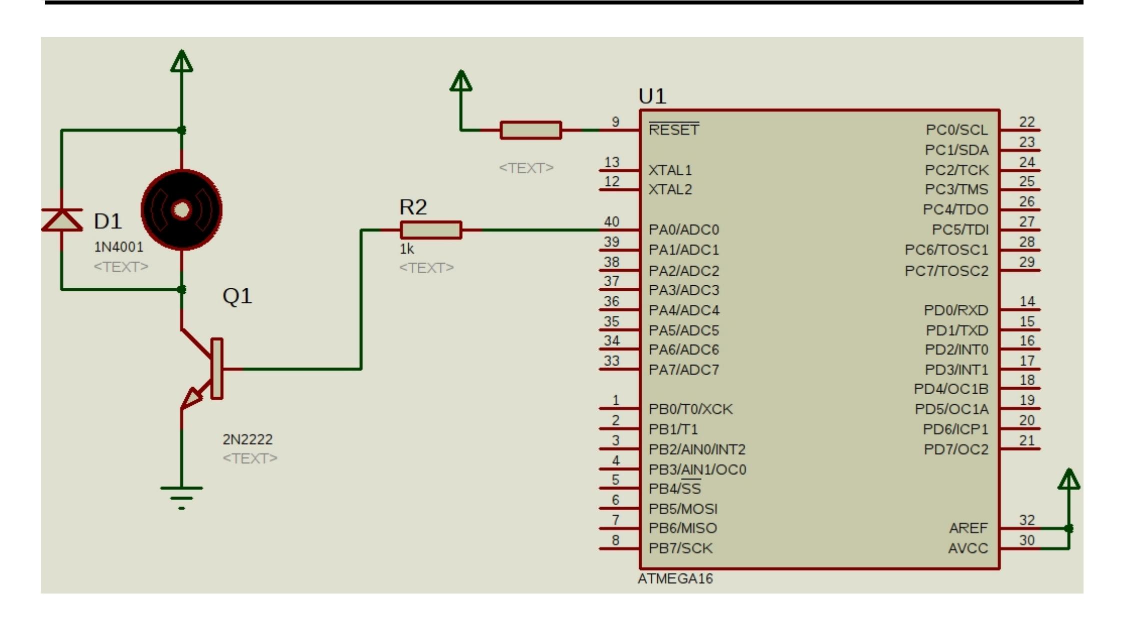
> ملاحظة: يتم استخدام مقاومة بين طرف المتحكم وقاعدة الترانزستور لأن أغلب الترانزستورات لا تتحمل تيار على طرف القاعدة أكبر من 10 مللي أمبير

تشغيل المُحركات DC

تعد المحركات (المواتير Motors) من اشهر العناصر الإلكتروميكانيكية والتي عادة ما نجدها في الأنظمة المدمجة الخاصة بالروبوتات، من أشهر هذه المحركات الـ DC motor (محرك التيار المستمر). هذا المحرك من المكونات التي تستهلك الكثير من التيار الكهربي لذا لا يمكن توصيله مباشرة بالمتحكم الدقيق ويجب أن نستخدم Driver circuit لتشغيله.

أبسط دائرة لتشغيل المحرك هي نفس الدائرة السابقة مع وجود بعض التعديلات البسيطة

> ملاحظة: يحتوي برنامج بروتس على محرك DC يسمى Active (Animated DC) وهذا هو الموجود في الصورة بالأسفل.



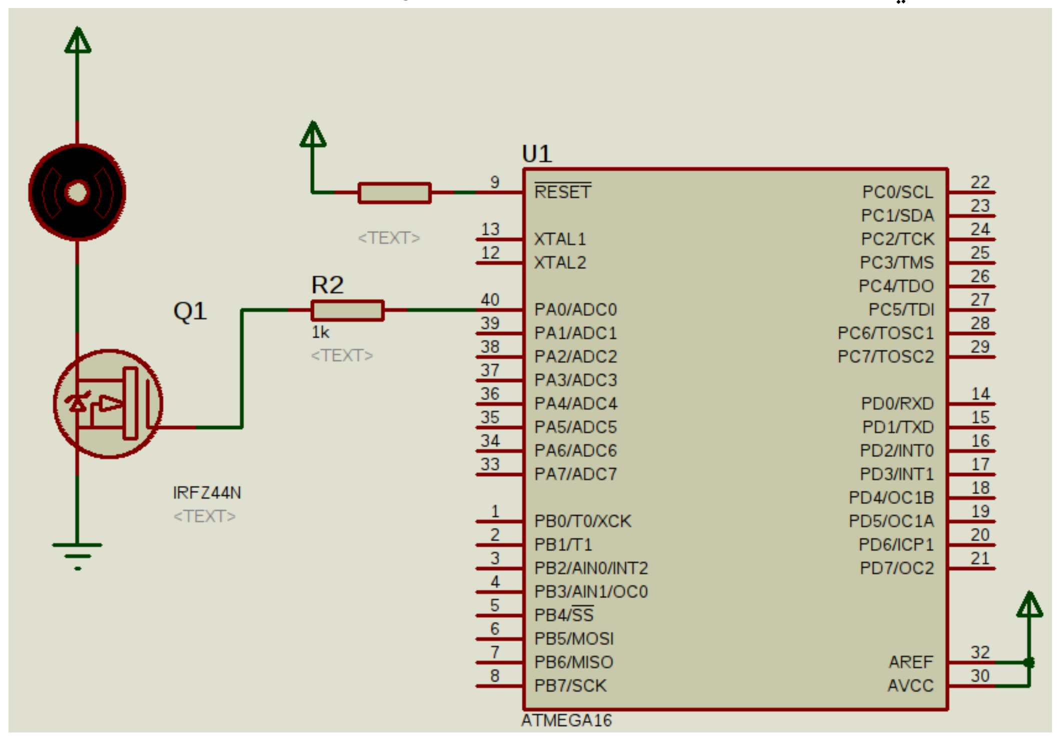
كما نرى من الصورة بالأعلى يتم توصيل المحرك في نفس مكان الدايودات الضوئية باستثناء وجود دايود 1n4001 متصل بطرفي المحرك، يستخدم هذا الدايود في الحفاظ على الترانزستور والمتحكم الدقيق من ظاهرة التيار المستحث العكسي.

هذه الظاهرة تحدث مع جميع الأحمال التي تتكون من ملف Coil مثل المحرك أو المُرحِل Relay وذلك بسبب أن الملفات النحاسية تستطيع أن تخزن بعض الطاقة بداخلها وعند قطع

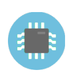
الكهرباء عنها فإنها تقوم بتفريغ هذه الطاقة المخزنة على هيئة تيار عكسي وقد يؤدي هذا التيار إلى تضرر الترانزستور والمتحكم الدقيق، ويتم استخدام الدايود بصورة معكوسة كما في الصورة السابقة لكي يمنع مرور هذه التيار من المحرك إلى الترانزستور

ملاحظة: في حالة استخدام ترانزستور NPN مثل الـ 2n2222 يستحسن أن تستخدم مقاومة على طرف المشع (قبل أن يتصل بطرف الـ GND) وذلك للحفاظ على الترانزستور من السخونة والاحتراق وهو ما يعرف باسم الـ thermal stability، حيث أن مرور التيار الخاص بتشغيل المحرك مباشرة إلى الترانزستور NPN بدون وجود هذه المقاومة قد يؤدي إلى ارتفاع درجة حرارة الترانزستور مع الوقت ثم يحترق.

بالرغم أن الدائرة السابقة تصلح لتشغيل المحرك إلا أنه يفضل استخدام أحد الترانزستورات الـ MOSFET بدل الـ NPN حيث أنها تستطيع تحمل التيار الكهربي العالي مثل الترانزستور IRZ44N (الذي يمكنه تحمل حتى 41 أمبير) كما هو موضح بالصورة التالية:



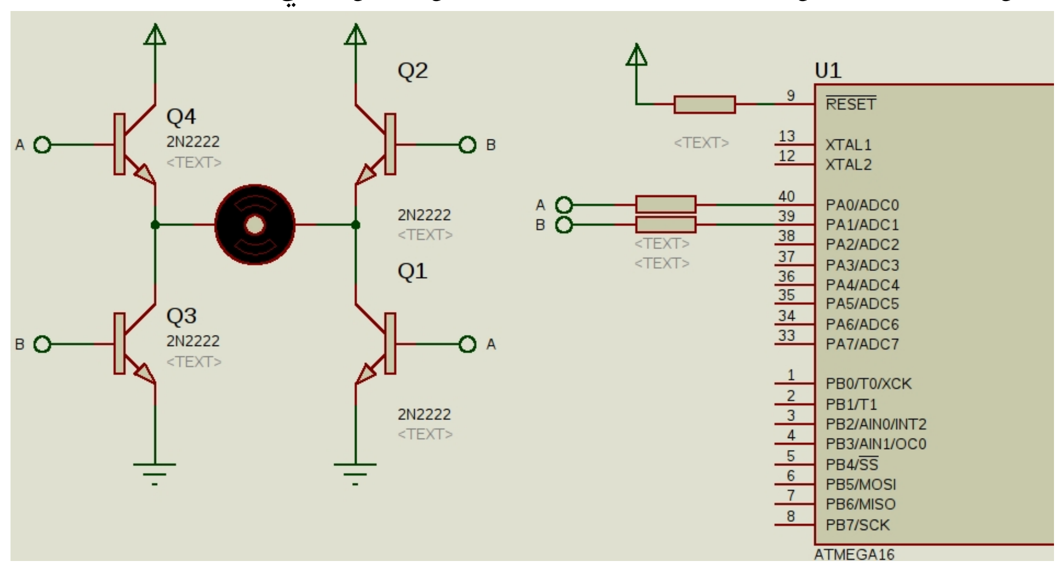
هذا الترانزستور يتميز بوجود دايود الحماية مدمج بداخله كما هو موضح بالصورة لذا لا داعي لإضافة الدايود 1n4001. يمكنك تجربة هذه الدائرة بنفس الكود الخاص بالمثال الأول.

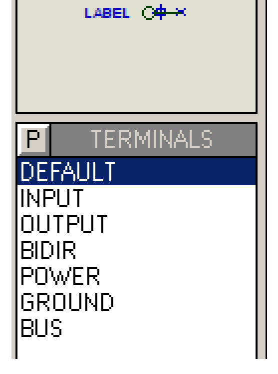


4.12 تشغيل المُحرك في كلا الاتجاهين

جميع التجارب السابقة لتشغيل المحرك بدائرة القيادة المعتمدة على الترانزستور كانت تعمل على تشغيل المحرك في اتجاه واحد فقط، ماذا إذا أردنا تشغيل المحرك في اتجاهين مختلفين؟

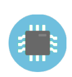
يمكننا ذلك باستخدام ما يعرف باسم H-Bridge وهي قنطرة مكونة من 4 ترانزستور على شكل الحرف H ويتصل المحرك بوسط هذه القنطرة مثل الشكل التالي:





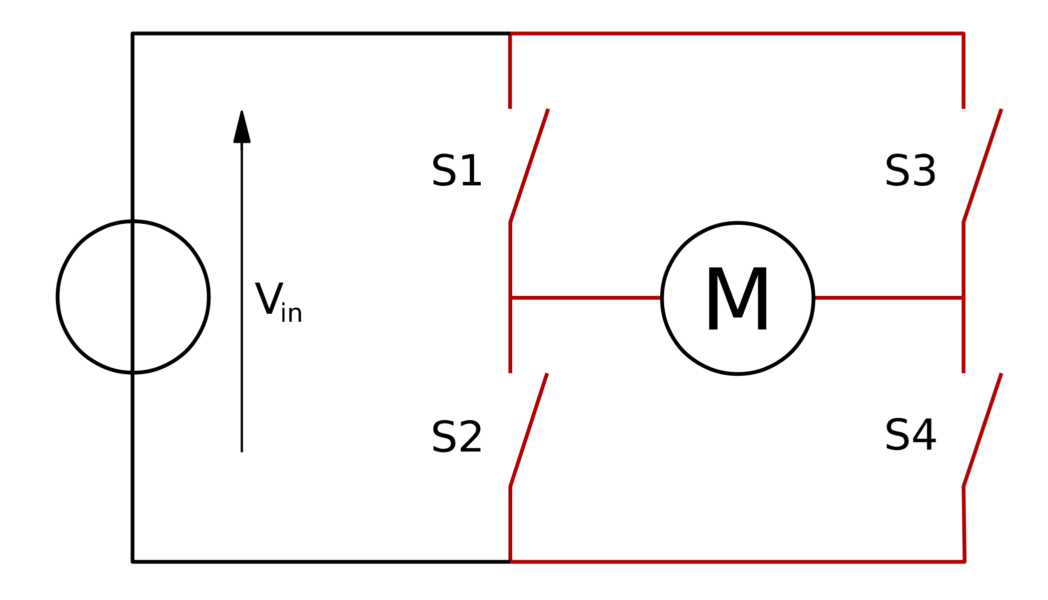
ملاحظة: الدوائر التي تحتوي على حرفي A,B تسمى Default Terminal وهي نقاط توصيل تستخدم لتوصيل المكونات ببعضها البعض بصورة سهلة بدلاً من الأسلاك المتداخلة ويمكنك الوصول إليها عبر الضغط على قائمة Terminal ثم اختيار Default.

لتوصيل أكثر من نقطة ببعضها البعض كل ما عليك فعله هو الضغط على هذه الدائرة وتسميها بحرف معين مثل A,B,C,D وستقوم هذه الدوائر بتوصيل جميع الدوائر المسماه بنفس الاسم ببعضها البعض.

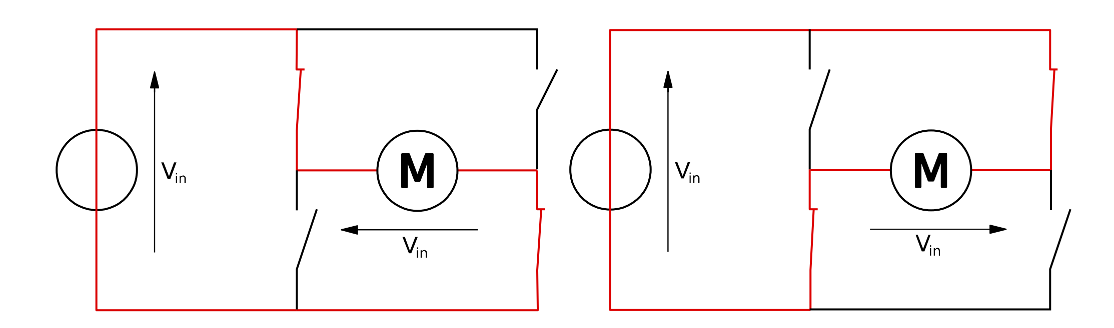


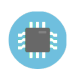
مبدأ عمل قنطرة H

تعمل الـ H-bridge مثل 4 مفاتيح، كما هو موضح في الصورة التالية



يعمل كل مفتاحين مع بعضهما البعض، فإما أن يتم إغلاق المفتاح S1 و S4 بحيث يمر التيار الكهربي من الطرف الأيسر إلى الطرف الأيمن للمحرك (بذلك يدور المحرك مع عقارب الساعة)، أو يتم إغلاق المفتاحين S2 و S3 بحيث يمر التيار الكهربي من الطرف الأيمن إلى الطرف الأيسر وبالتالي يدور المحرك عكس عقارب الساعة.





البرنامج

لتشغيل هذه القنطرة يتم استخدام طرفين مثل PA0 و PA1 حيث يتصل كل طرف بعدد 2 من الترانزستورات كما هو موضح في الصورة الأولى، ويتم التحكم باتجاه دوران المحرك عن طريق تشغيل الطرف PA0 وإطفاء PA1 أو العكس (لتغير اتجاه دوران المحرك).

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{

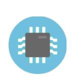
        DDRA = 0b11111111;

while(1)
 {
        PORTA = 0b00000001;        // تشغيل المحرك مع عقارب الساعة
        _delay_ms(2000);           // انتظر لمدة ثانيتين
        PORTA = 0b00000010;        // تشغيل المحرك عكس عقارب الساعة
        _delay_ms(2000);           // انتظر لمدة ثانيتين
        PORTA = 0b00000000;        // إطفاء جميع الترانزستور وإيقاف المحرك
        _delay_ms(2000);           // انتظر لمدة ثانيتين
 }
   return 0;
}
```

الفصل الخامس

" قل للذي أحصى السنين مفاخرا يا صاح ليس السرّ في السنوات
لكنه في المرء كيف يعيشها في يقظة ، أم في عميق سبات "

إيليا أبو ماضي – شاعر عربي



5. قواعد لغة السي للأنظمة المدمجة



For Embedded Systems

في هذا الفصل سنتعلم بعض القواعد والصيغ الشهير للغة السي المعيارية والمستخدمة بشكل كبير في تطوير الأنظمة المدمجة. تتميز الصيغ المعيارية بإمكانية تطبيقها على مختلف المُتحكِمات الدقيقة طالماً أن المترجم الخاص بها يدعم لغة السي.

- ✔ أنواع البيانات في الأنظمة المدمجة Data-types
- ✔ العمليات الحسابية واختصاراتها Arithmatic operations
- ✔ العمليات المنطقية واختصاراتها Logic operations
- ✔ عمليات الأزاحة
- ✔ التلاعب بالبتتات

5.1 أنواع البيانات في الأنظمة المدمجة Data-types

في عالم الأنظمة المدمجة هناك معيار مختلف قليلاً لتعريف البيانات مثل المتغيرات والثوابت. حيث نجد أن معظم المُتحكِمات الدقيقة خاصة من فئة 8-Bit تمتلك قدر محدود جداً من الذاكرة سواء الـ ROM أو الـ RAM (بعض المُتحكِمات قد يمتلك 128 بايت فقط من الـ RAM). مما يتطلب أسلوب فعال لكتابة أكواد تحقق الهدف المطلوب **بأقل استهلاك للذاكرة**.

مثلاً في حالة كتابة برنامج بلغة السي على الحاسب الآلي التقليدي والذي يمتلك معالج وذاكرة ضخمة (تقدر بالجيجا بايت)وكلاهما مخصص للتعامل مع البيانات 32 أو 64 بت، سنجد أن معظم المتغيرات الرقمية تكون int أو double أو float أما في حالة الأنظمة المدمجة يعد استخدام هذه الأنواع "اهدار للذاكرة" لذا سنتعلم في هذا الجزء كيفية برمجة أنواع البيانات المختلفة بصورة مُحسنة خصيصاً لذاكرة المُتحكِمات الدقيقة.

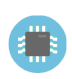
البيانات الرقمية

تعتبر الأرقام هي أكثر أنواع البيانات استخداماً في عالم الأنظمة المدمجة فهي تعبر عن قيم المتغيرات مثل قراءات الحساسات (حرارة، ضغط، رطوبة، ضوء، ..الخ) أو قيم المخارج مثل قيمة أي PORT أو سرعة Motor أو توقيت زمني .. إلخ. وتعتبر الأرقام هي أكثر ما يستهلك الذاكرة (خاصة الـ SRAM) لذا سيكون لها نصيب كبير من الشرح في هذا الفصل.

أنواع البيانات التقليدية

هذه القائمة تمثل أشهر أنواع البيانات المستخدمة في تعريف المتغيرات في لغة السي مع حجم الذاكرة الذي تستهلكه كل منها. مع العلم أنه يمكنك برمجة جميع الأنظمة المدمجة بها.

> أنواع البيانات التي تستخدم في حفظ الأرقام والموجبة والسالبة تسمى signed numbers
> أما إذا وضعنا قبلها كلمة unsigned فإنها تصلح للتعامل مع الأرقام **الموجبة فقط** ولكن بضعف مساحة التخزين



نوع البيانات	استهلاك الذاكرة (عدد Bytes)	أقصى قيمة يمكن وضعها داخل المتغير
int	2-bytes (16 bit)	-32,768 → +32,767
unsigned int	2-bytes (16 bit)	65,535
float	4-bytes (32 bit)	1.2E-38 to 3.4E+38
double	8-bytes (32 bit)	2.3E-308 to 1.7E+308
long	4-bytes (32 bit)	-2,147,483,648 → +2,147,483,647
unsigned long	4-bytes (32 bit)	4,294,967,295
long long	8-bytes (64 bit)	$- 9.223372037 \times 10^{18}$ $+ 9.223372037 \times 10^{18}$
unsigned long long	8-bytes (64 bit)	$1.844674407 \times 10^{19}$

ملاحظة: نوعي البيانات float و double دائماً يكونا من نوع Signed ولا يمكن استخدام العلامة unsigned معهما

كما نلاحظ من الجدول السابق أن جميع أنواع البيانات الشهيرة تستهلك مساحة تبدأ من 2 بايت حتى 8 بايت وتعتبر هذه المساحة كبيرة نسبياً في عالم الأنظمة المدمجة. فمثلاً قيمة أي مُسجِل مثل PORTx أو PINx أو DDRx لن تزيد أبداً عن 8 بت.

والآن لنتخيل أننا نحتاج متغير اسمه ButtonStatus من نوع int و سيستخدم هذا المتغير في حفظ قراءة أطراف البورت B وبما أن المُسجِل PINB من نوع 8 بت إذا كل ما يتطلبه هو متغير بسعة 8 بت فقط ولا داعي أبداً لاستخدام متغير بمساحة 16 بت. لذا فجميع أنواع البيانات السابقة تعتبر اهدار للذاكرة.

لحل هذه المشكلة يتم استخدام أنواع البيانات المعيارية ANSI C - C99 وهي صورة محسنة لأنواع البيانات وتمكننا من اختيار قيمة المتغيرات بالدقة والطول المطلوب دون أي اهدار للذاكرة.

Signed	أقصى قيمة	Unsigned	أقصى قيمة
int8_t	-128 → +127	**uint8_t**	0 → 255
int16_t	-32,768 → +32,767	**uint16_t**	0 → 65,535
int32_t	-2,147,483,648 → +2,147,483,647	**uint32_t**	0 → 4,294,967,295
int64_t	$- 9.223372037 \times 10^{18}$ $+ 9.223372037 \times 10^{18}$	**uint64_t**	$1.844674407 \times 10^{19}$

يستهلك كل متغير من القائمة السابقة مساحة = الرقم المكتوب بعد كلمة **int** أو **uint** فمثلاً int8_t تعني أن هذا المتغير من نوع int ويصلح لتخزين الأرقام **الموجبة والسالبة** في مساحة تخزينية = 8 بت فقط (1 بايت). بينما uint16_t يعني أن هذا المتغير من نوع unsigned int ويصلح لتخزين الأرقام الموجبة فقط بمساحة تصل إلى 16 بت (2 بايت).
والآن لنأخذ مجموعة من الأمثلة:

ما هو نوع المتغير المطلوب للقراءة أو الكتابة في أي بورت (مثل port A, B,C)؟

- بما أن جميع البورتات يتم قراءتها من خلال المُسجِل PINx والذي يكون 8 بت وكذالك يتم كتابة في أي بورت باستخدام المُسجِل PORTx والذي يعتبر 8 بت أيضاً إذا أفضل نوع بيانات هو **uint8_t** حيث يمكنه تخزين الأرقام من 0b0000000 إلى 0b11111111

ما هو المتغير المناسب لتخزين عداد رقمي من صفر إلى 10,000؟

- بما أن الرقم لا يحتوي على أرقام سالبة وأقصى رقم مطلوب هو 10000 وهذا الرقم أكبر من 255 لذا لا يمكن استخدام متغير بمساحة 8 بت ويجب أن نستخدم متغير بمساحة 16 بت والذي يستطيع التعامل مع ارقام تصل إلى 65,535. إذا أفضل نوع بيانات هو **uint16_t**

ما هو المتغير المناسب لتخزين درجة حرارة تتراوح بين سالب 50 إلى موجب 100؟

- هناك إجابتين لهذا السؤال، الأولى: في حالة أننا نريد تخزين أرقام صحيحة فقط مثل 20 أو 50 أو أي رقم بدون كسر عشري فسنجد أن افضل نوع بيانات هو **int8_t** حيث يستطيع هذا النوع من البيانات أن يخزن أي قيمة تتراوح بين -128 إلى + 127 ويستهلك 1 بايت فقط من مساحة الذاكرة.
- الثانية: في حالة وجود كسر عشري مثل أن 25.1 أو 30.5 يجب أن نستخدم نوع البيانات **float** مع العلم أنه في المقابل سيتم استهلاك مساحة ذاكرة = 4 بايت (32 بت).

السؤال الأخير يدفعنا لنقطة هامة جداً وتسمى الـ Variable precision - دقة المتغير – بعض المتغيرات يجب أن يتم تخزينها بدقة عالية جداً. والبعض الآخر لا يتطلب هذه الدقة فمثلاً إذا طُلب منك أن تصنع ساعة رقمية وبها حساس لعرض درجة حرارة الغرفة، سنجد أن المتغير

المناسب لتخزين قيم هذا الحساس هو int8_t وذلك لأن درجة الحرارة المتوقعة لأي غرفة لن تزيد عن 45 أو تقل عن -5 (حتى أقصى بلاد العالم برودة تكون درجة حرارة المنازل معزولة عن درجة الهواء الخارجي بفارق 10 درجة على الأقل) كما أن الشخص الذي سيستخدم هذه الساعة الرقمية لن يهتم بعرض درجة الحرارة بدقة 25.20 وسيكتفي فقط بمعرفة الرقم الصحيح (مثل 25 درجة). ولكن إذا طلب منك تصميم مقياس حرارة طبي مثل المستخدم في المستشفيات لقياس درجة حرارة المرضى فيجب أن تكون درجة الحرارة محددة بدقة مثل 36.25 لذا يجب أن يتم استخدام متغير من نوع float لتخزين ومعالجة هذه القيم.

أمثلة برمجية

قم بتوصيل 8 دايودات ضوئية على أطراف البورت A ثم اكتب برنامج يقوم بإخراج القيم من 0 إلى 255 على البورت مع وضع تأخير زمني بين كل قيمة 100 مللي ثانية.

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <avr/delay.h>

int main(void)
{
 uint8_t counter;                 // متغير بمساحة 8 بت لتخزين قيمة العداد

 DDRA = 0b11111111;               // ضبط كل أطراف البورت لتعمل كخرج

 for (counter = 0; counter <= 255; counter++)        // عداد من 0 إلى 255
        {
                PORTA = counter;
                _delay_ms(100);
        }

   return 0;
}
```

صور مختلفة لكتابة المتغيرات الرقمية

في لغة السي يمكن كتابة قيم المتغيرات الرقمية بطرق متنوعة. فمثلاً في الأمثلة السابقة قمنا باستخدام الصيغة الرقمية الثنائية Binary مثل:

```
uint8_t variable = 0b00000011;
```

أيضاً يمكن كتابة المتغيرات بالصيغة العشرية Decimal وهي نفس صيغة الأرقام التي نستخدمها في حياتنا اليومية فمثلاً الرقم 0b00000011 يساوي رقم 3 Decimal ويمكن كتابة نفس الأمر السابق بالصورة التالية:

```
uint8_t variable = 3;
```

ويمكن استخدام الصيغة الستة-عشرية Hexadecimal. فمثلاً لنفترض أننا نريد تعيين المتغير variable بقيمة 0b11110000 والتي تساوي رقم 120 بالصيغة العشرية

```
uint8_t variable = 0b11110000;          // Binary
uint8_t variable = 120;                 // Decimal
uint8_t variable = 0xF0;                // Hexadecimal
```

مرفق مع الكتاب ملحق خاص بنظام الأعداد وتحويل قيم الأرقام من وإلى الصيغ الثنائية والعشرية والستة-عشرية.
أيضاً لاحظ أن القيم الستة-عشرية تبدأ بـ **0x** بينما الصيغة الثنائية تبدأ بـ **ob**

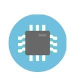
مراجع إضافية

http://www.mjma3.com/programming/basics.html

فيديو: تحويل الأعداد من النظام العشري إلى الثنائي
https://www.youtube.com/watch?v=-yrwpEuRjl0

فيديو: تحويل الأعداد من النظام الثنائي إلى العشري
https://www.youtube.com/watch?v=lLbc5_JpTWI

فيديو: تحويل الأعداد من النظام الثنائي إلى الست-عشري
https://www.youtube.com/watch?v=B33Iof0bUKg



5.2 العمليات الحسابية Arithmetic Operations

تستطيع مترجمات لغة السي المعيارية المخصصة للمتحكمات الدقيقة أن تتعامل مع نفس أوامر العمليات الحسابية التي يمكن تنفيذها على الحواسيب مثل عمليات الجمع والطرح والضرب والقسمة. لنأخذ بعض الأمثلة:

```
int X, Y, Z;          //X, Y, Z تعريف 3 متغيرات
X = 5 + 6;            // X عملية جميع رقمين ووضع الناتج في المتغير
Y = 50 - x;           // Y عملية طرح رقم من متغير ووضع الناتج في المتغير
Z = X * Y;            // Z عملية ضرب متغيرين ووضع النتيجة في المتغير
```

أيضاً يمكن تنفيذ العمليات الحسابية على قيمة المتغير نفسه. فمثلاً قيمة المتغير x في المثال السابق = 11 والآن نريد أن نزيد عليها 4 لتصبح 15. يمكن تنفيذ ذلك بالطريقة التالية

```
X = X+4;              // 15 الآن قيمة المتغير تساوي
```

الأمر السابق يعني: قم بإضافة الرقم 4 إلى القيمة الموجودة بالفعل في المتغير X ثم قم بكتابة النتيجة مرة أخرى داخل المتغير X وبذلك تصبح قيمته 15. مع العلم أنه يمكن إجراء نفس الطريقة مع جميع العمليات الحسابية

```
X = X - 2;            // أطرح الرقم 2 من المتغير
X = X * 4;            // 4 اضرب قيمة المتغير في
```

الصور المختصرة

يمكن إجراء العمليات الحسابية على المتغيرات بصورة مختصرة. فمثلاً إذا أردنا جميع الرقم 4 على قيمة المتغير X فيمكن ذلك بأسلوب مختصر عبر إضافة علامة الجمع قبل أشارة =

```
X += 4;               // X = X+4 اختصار الأمر
```

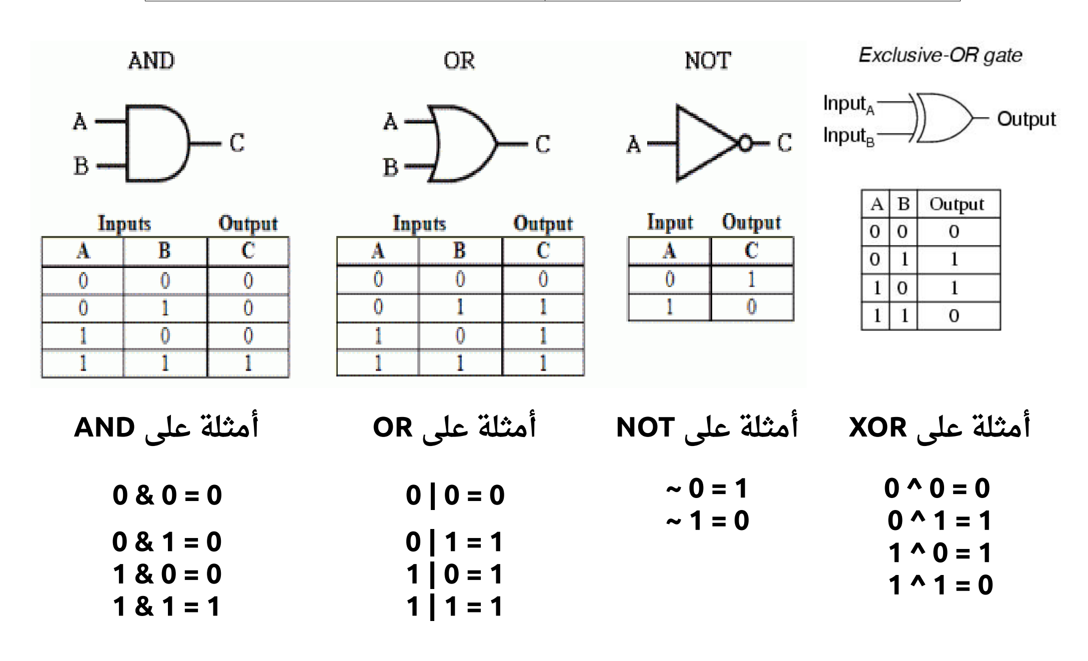
أيضاً يمكن تطبيق نفس الاختصار على أي عملية حسابية

```
X -= 4;               // X = X - 4 اختصار الأمر
X *= 4;               // X = X * 4 اختصار الأمر
X /= 2;               // X = X / 2 اختصار الأمر
```

5.3 العمليات المنطقية Logic Operation

الـ Logic operation مجموعة من الأوامر المنطقية مثل AND, OR, XOR, Invert, Shift هذه الأوامر تساعدنا على القيام ببعض التحكم المتقدم سواء على المتغيرات أو إعدادات المُتحكِم أو أطراف المُتحكِم (الدخل أو الخرج). جميع العمليات المنطقية (تمسى أيضاً الدوال المنطقية Logic Function) تمثل نفس البوابات المنطقية المستخدمة في الدوائر الإلكترونية لكن يتم تطبيقها بصورة برمجية. الجدول التالي يوضح قائمة بهذه العمليات والرموز المستخدمة في لغة السي لتنفيذ كل واحدة منهم.

Logic Function name	Logic Function in "ANSI C"
OR	|
AND	&
NOT	~
XOR	^
Shift Left	<<
Shift Right	>>



AND:

Inputs		Output
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

OR:

Inputs		Output
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

NOT:

Input	Output
A	C
0	1
1	0

XOR:

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

أمثلة على AND

0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1

أمثلة على OR

0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1

أمثلة على NOT

~ 0 = 1
~ 1 = 0

أمثلة على XOR

0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0

لنفهم هذه العمليات بصورة أفضل لنأخذ مجموعة من الأمثلة البرمجية.

استخدام NOT - يُستخدَم الأمر NOT “invert” value في عكس جميع البتات داخل أي متغير أو أي مُسجِل فمثلاً إذا كان لدينا المتغير Z كالتالي:

Example 1:
```
unit8_t Z = 0b00000000;
Z = ~ Z;                    // invert all Z bits (Z = 0b11111111)
```
تطبيق الأمر الأخير يعني: قم بعكس جميع البتات الموجودة في المتغير X ثم ضع القيمة الجديدة داخل المتغير X وبذلك تصبح القيمة = 0b11111111

Example 2:
```
unit8_t Z = 0b11110000;
Z = ~ Z;                    // invert all Z bits (Z = 0b00001111)
```

Example 3:
```
unit8_t Z = 0b00110011;
Z = ~ Z;                    // invert all Z bits (Z = 0b11001100)
```

استخدام OR - لنفترض أن لدينا متغير 8 بت xNumber وقيمته = 12 (بالنظام العشري) أو 0b00011000 بالنظام الرقمي كما هو موضح:

Example 4:
```
uint8_t xNumber = 0b00011000;
xNumber = xNumber | 0b00000001;          // 0b000011000 OR 0b0000001
```

الأمر السابق يعني قم بتنفيذ العملية المنطقية OR مع محتوى المتغير xNumber ثم ضع النتيجة داخل المتغير xNumber مما سيجعل قيمة المتغير تصبح 0b00011001. وذلك لأن الأمر OR يقوم بعمل OR operation على كل بت بين الرقمين كالتالي:

قيمة **xNumber** الأصلية	0	0	1	1	0	0	0	0
الرقم الثاني	0	0	0	0	0	0	0	1
نتيجة الـ OR	0	0	**1**	**1**	0	0	0	**1**

Example 5:
```
xNumber = 0b11111000;
xNumber = xNumber | 0b00000011;
```

عند تنفيذ الأمر OR ستكون قيمة المتغير= 0b11111011 كما هو موضح في الجدول التالي:

قيمة **xNumber** الأصلية	1	1	1	1	1	0	0	0
الرقم الثاني	0	0	0	0	0	0	1	1
نتيجة الـ OR	**1**	**1**	**1**	**1**	**1**	0	**1**	**1**

استخدام AND - تعمل عملية الـ AND مثل ضرب رقمين. وعند تطبيقها بين المتغيرات يتم عمل AND بين كل بت في المتغير الأول وما يقابلها في المتغير الثاني

```
Example 6:
```

;xNumber = 0b11111000

// 0b11111000 AND 0b1000001 xNumber = xNumber & 0b10000001;

النتيجة ستكون 0b10000000 وذلك لأن نتيجة عمل AND مع أي بت تحتوي على صفر = صفر لذا نجد البت الأخير فقط هي التي ستظل 1 لأن 1 & 1 = 1

قيمة **xNumber** الأصلية	1	1	1	1	1	0	0	0
الرقم الثاني	1	0	0	0	0	0	0	1
نتيجة الـ AND	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

اختصار العمليات المنطقية

يمكن اختصار العمليات المنطقية مثل العمليات الحسابية وذلك عبر وضع الأمر المنطقي قبل علامة = فمثلاً يمكن إجراء الـ logic operations كالتالي:

```
xNumber   |=  0b00000001;                  // xNumber = xNumber | 0b00000001;
xNumber  &=  0b00000001;                   // xNumber = xNumber & 0b00000001;
xNumber  ^=  0b00000001;                   // xNumber = xNumber ^ 0b00000001;
```

ملاحظة: يمكن اختصار جميع العمليات المنطقية ماعدا عملية (العكس Invert) والتي تكتب فقط بالصورة التالية:

xNumber = ~ xNumber

5.4 عمليات الإزاحة Shift operations

تعرف عمليات الأزاحة بأنها تحريك البتات إلى اليمين أو اليسار داخل متغير أو مُسجِل أو أي قيمة رقمية. تعد هذه العمليات من أهم الأوامر التي تستخدم في برمجة المعالجات والمُتحكِمات الدقيقة كما سنرى.

مثال: لنفرض أن لدينا الرقم 0b00000001 لنقم بتطبيق عمليات الإزاحة لجهة اليسار.

الرقم الأصلي	0b0000000**1**
إزاحة بمقدار 1 بت جهة اليسار <<	0b000000**1**0
إزاحة بمقدار 2 بت جهة اليسار <<	0b00000**1**00
إزاحة بمقدار 3 بت جهة اليسار <<	0b0000**1**000

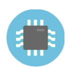
مثال: لنفرض أن لدينا الرقم 0b00000101 لنقم بتطبيق نفس العمليات السابقة عليه.

الرقم الأصلي	0b00000**101**
إزاحة بمقدار 1 بت جهة اليسار <<	0b0000**101**0
إزاحة بمقدار 2 بت جهة اليسار <<	0b000**101**00
إزاحة بمقدار 3 بت جهة اليسار <<	0b00**101**000

مثال: لنفرض أن لدينا 0b01100000 لنقم بتطبيق عمليات الإزاحة لجهة اليمين.

الرقم الأصلي	0b0**11**00000
>> إزاحة بمقدار 1 بت جهة اليمين	0b00**11**0000
>> إزاحة بمقدار 1 بت جهة اليمين	0b000**11**000
>> إزاحة بمقدار 1 بت جهة اليمين	0b0000**11**00

تدعم لغة السي الأوامر البرمجية لعمل الإزاحة لجهة اليمين أو اليسار وتكتب كالتالي:



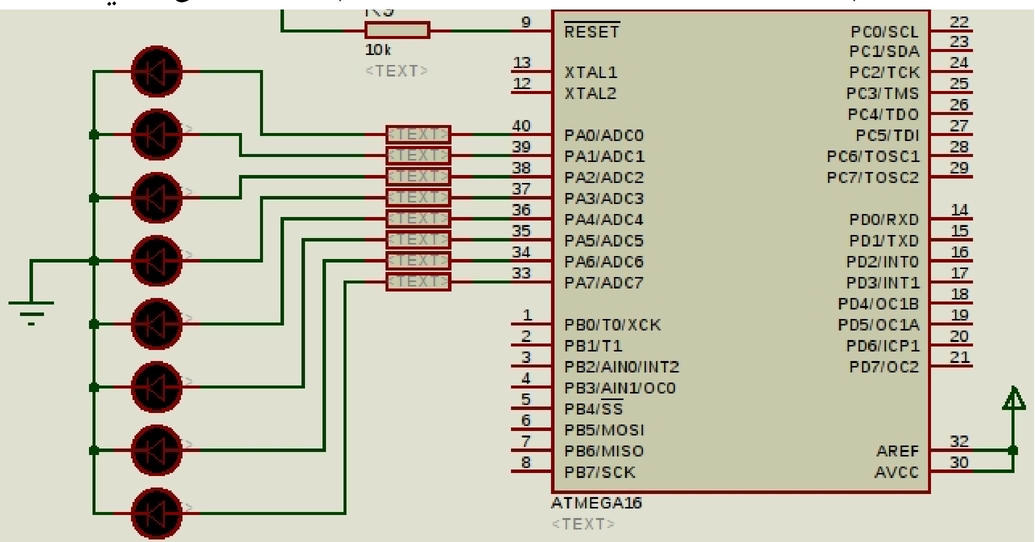
الإزاحة لجهة اليسار:

(قيمة الإزاحة >> الرقم)

الإزاحة لجهة اليمين:

(قيمة الإزاحة << الرقم)

المثال الأول: قم بتوصيل 8 دايودات ضوئية على البورت A ثم جرب البرنامج التالي:

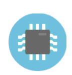



```
int main(void)
{
        uint8_t counter;
        DDRA = 0xff;

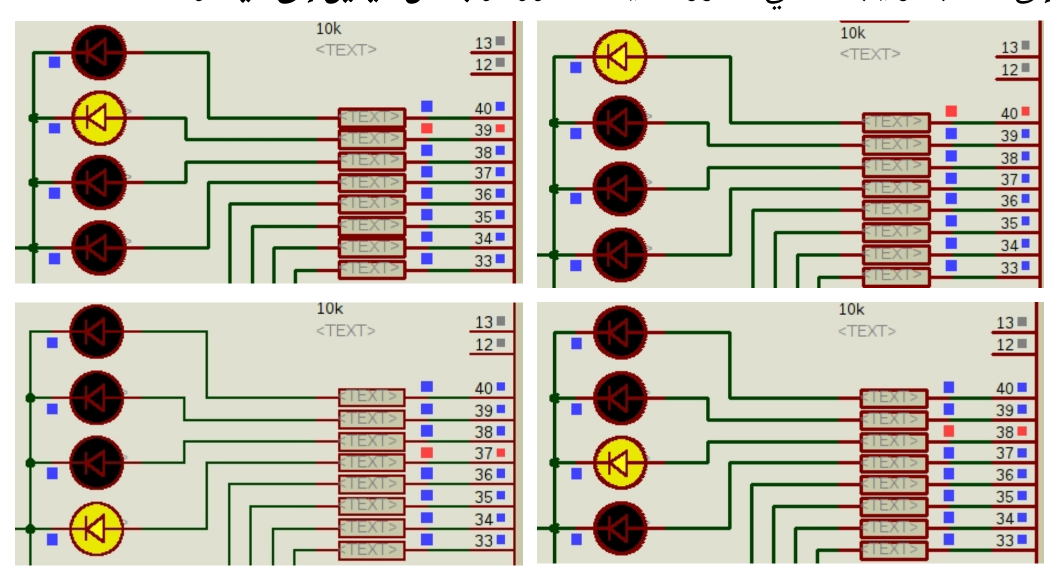
while(1)
 {

  for(counter = 0; counter <= 7; counter++)
        {
            PORTA = (1 << counter);
           _delay_ms(500);
        }
 }

  return 0;
}
```



نتيجة تنفيذ الكود ان الدايودات الضوئية ستبدأ بالاضاءة واحدة تلو الأخرى بداية من PA0 إلى PA7 بالترتيب كما في الصور التالية **(الصور مرتبة من اليمين إلى اليسار):**



شرح الكود

في البداية تم تعريف متغير يسمى counter حيث ستسخدم هذا المتغير في زيادة قيمة الإزاحة بصورة تلقائية. أيضاً تم تعريف جميع أطراف البورت A لتعمل كخرج.

```
uint8_t counter;
DDRA = 0xff;
```

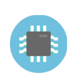
داخل الـ while loop استخدمنا دالة التكرار for وذلك لزيادة قيمة المتغير counter بصورة تلقائية من 0 إلى 7.

```
for(counter = 0; counter <= 7; counter++)
```

والآن يأتي أمر المحاذاة. حيث تم استخدام البورت A لعرض كيف تتحرك البت 1 لجهة اليسار مع تأخير زمني نصف ثانية وهذا ما يسبب أن تضيء الدايودات الضوئية بالترتيب واحدة تلو الأخرى.

```
PORTA = (1 << counter);
_delay_ms(500);
```

لاحظ أن الرقم 1(المكتوب بالصيغة العشرية) الموجود في العبارة PORTA = (1 << counter) يوازي بالضبط PORTA = (0b00000001 << counter)



بسبب الدالة for نجد أنه عندما تكون قيمة counter = 1 فهذا يجعل الأمر السابق يساوي

```
PORTA = (0b00000001 << 1);
```

وعندما تكون قيمة counter = 1 فهذا يوازي الأمر

```
PORTA = (0b00000001 << 2);
```

وعندما تكون قيمة counter = 3 فهذا يوازي الأمر

```
PORTA = (0b00000001 << 3);
```

وهكذا ... وتستمر هذه الدورة إلى ما لا نهاية.

5.5 التحكم على مستوى البت الواحدة Single Bit

في الفصل السابق قمنا بالتعامل مع بعض المُسجِلات مثل PINx, PORTx, DDRx وذلك بكتابة بعض القيم الرقمية داخل هذه المُسجِلات. جميع الأمثلة السابقة اشتركت في أمر هام وهو كتابة جميع قيم البتات داخل كل مُسجِل في نفس الوقت. فمثلاً عندما نقوم بضبط أول 3 أطراف للبورت A لتعمل كخرج فأننا نكتب DDRA = 0b00000111 **<u>مما يجعل الأطراف الثلاث الأولى خرج والأطراف الباقية تعمل كدخل.</u>**

لكن ماذا إذا أردنا تعديل طرف واحد (وليكن الطرف الأول فقط PA0) وفي نفس الوقت لا نريد أن نغير قيمة أو محتوى أي بت أخرى في المُسجِل DDRA؟

يمكننا ذلك بسهولة عبر دمج مجموعة من العمليات المنطقة وعمليات الأزاحة. حيث تتوفر صيغ برمجية معينة تمكننا من وضع 1 أو 0 داخل بت محددة في أي متغير أو مُسجِل. كما يمكننا أن نعكس بت معينه أو نعزلها عن باقي البتات.

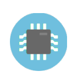
كتابة 1 داخل أي بت Set Bit

تعرف الصيغة البرمجية Set Bit أنها وضع 1 داخل قيمة أي بت. بالعودة للسؤال السابق لنفترض أننا نريد ضبط الطرف PA0 فقط ليعمل كخرج بغض النظر عن باقي أطراف البورت A. يمكن ذلك بسهولة عبر دمج الأمر OR مع left shift وتكون الصيغة العامة

```
Register |= (1 <<  bitName);
```

يتم استبدال كلمة Register بالمسجل المطلوب ويتم استبدال كلمة bitName باسم البت المطلوب تغيير قيمتها. فمثلاً بتطبيق هذه الصيغة على المُسجِل DDRA و الطرف PA0 نكتب:

```
DDRA |= (1<< PA0);            // set PA0 to work as output
```



أيضاً يمكننا استخدام نفس الصيغة لتشغيل نفس البت من المُسجِل PORTA

```
PORTA |= (1<<PA0);           // turn on PA0
```

كذلك يمكننا استخدام العملية OR لدمج أكثر من صيغة من النوع السابق في أمر واحد، فمثلاً لنفترض أننا نريد ضبط الطرف PA1 و PA2 ليعملان كخرج، يمكننا إما أن نكتب:

```
DDRA |= (1<<PA1);
DDRA |= (1<<PA2);
```

أو يمكن اختصار الأمرين في جملة واحدة فقط كالتالي:

```
DDRA |= (1<<PA1) | (1<<PA2);
```

كتابة 0 داخل أي بت Reset Bit

عملية الـ Reset Bit هي عكس عملية Set Bit وتعني وضع صفر داخل أي بت مطلوبة وتتم عبر دمج 3 أوامر AND – left shift – invert وتكون الصيغة القياسية كالتالي:

```
Resgister &= ~ (1 << bitName);
```

مثال: لنفترض أننا نريد جعل الطرف PC3 يعمل كدخل بغض النظر عن باقي أطراف البورت C

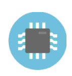
```
DDRC &= ~(1<<PC3);
```

أيضاً يمكن استخدام نفس الصيغة في إلغاء تشغيل أي طرف، فمثلاً إذا أردنا تشغيل دايود ضوئي على البت PA0 لمدة ثانية واطفاءة لمدة ثانية فيمكننا أن نكتب:

```
int main() {

 DDRA |= (1<<PA0);                    // set 1 in PA0 (output)
 while (1)
  {
        PORTA |= (1<<PA0);            // turn on led (set PA0)
        delay_ms(1000);

        PORTA &= ~ (1<<PA0);          // turn off led (reset PA0)
        delay_ms(1000);
   }
 return 0;
}
```



عكس بت محددة Toggle Bit

تعد الصيغة البرمجة Toggle Bit من أفضل الصيغ المستخدمة في البرمجة وتعني أنه في حالة أن البت المطلوبة = 1 قم بعكسها إلى 0 وإذا كانت تساوي 0 قم بعكسها إلى 1. وتكون الصيغة القياسية كالتالي:

```
Resgister ^= (1 << BitName);
```

باستخدام هذه الصيغة يمكننا اختصار المثال blinking led بأمرين فقط (بدلاً من 4 أوامر).

```
while (1)
{
	PORTA ^= (1<<PA0);
	_delay_ms(1000);
}
```

تعد مجموعة الصيغ البرمجية السابقة من أهم الأوامر التي قد تستخدمها في برمجة النظم المدمجة خاصة أنها مكتوبة بلغة السي المعيارية مما يجعلها طريقة "معيارية" لعمل set أو reset لأي بت مهما أختلف المُتحكِم الدقيق أو الشركة المنتجة (طالماً أن المترجم الخاص بهذا المُتحكِم يدعم لغة السي المعيارية).

5.6 القراءة من بت واحدة Read single bit

في الفصل السابق شاهدنا بعض الأمثلة عن قراءة مفتاح إلكتروني push button(switch) وذلك عبر قراءة بورت كامل من خلال المُسجِل PINx. الطريقة السابقة كانت تقرأ جميع بتات البورت PINx وتقارنها برقم مثل:

```
	if (PINB == 0b00000001)
		{PORTA = 0b00000001;}
	else
		{PORTA = 0b00000000;}
```

لكن هناك صيغة أفضل في حالة الرغبة بقراءة طرف واحد فقط وهي كالتالي:

Reister & (1<<bitName)

يمكن تطبيق نفس الصيغة على الكود السابق ليصبح كالتالي:

```
	if (PINB & (1<<PA0))		//تعني لو أن قيمة الطرف الأول تساوي 1
		{PORTA = 0b00000001;}
```

الفصل السادس

"النجاح ليس نتيجة لعدم ارتكاب أي أخطاء،
ولكنه نتيجة لعدم تكرار نفس الخطأ مرتين "

جورج برنارد شو - مؤلف أيرلندي شهير.

6. الفيوزات، الحماية، الطاقة وسرعة التشغيل



هذا الفصل يشرح الإعدادات المتقدمة لمتحكمات AVR مثل مفهوم الفيوزات ووظائفها المختلفة مثل تغير سرعة التشغيل Clock Rate واستهلاك الطاقة، حماية البرامج الموجودة على المُتحكِم من السرقة أو التعديل وتشغيل بعض الخصائص المتقدمة الأخرى.

- ✔ الفيوزات Fuses للمُتحكِم ATmega16
- ✔ تغير سرعة (تردد) المُتحكِمات
- ✔ بتات الإغلاق والحماية Lockbits
- ✔ برمجة الفيوزات
- ✔ استهلاك الطاقة والعمل على البطاريات

Fuses & Lockbits 6.1

في جميع المُتحكِمات الدقيقة يجب أن تمتلك طريقة ما لتتحكم في بعض الإعدادات بصورة دائمة لا تتغير مع انقطاع الكهرباء عن المُتحكِم الدقيق، فمثلا مُتحكِمات الـ AVR تأتي بصورة افتراضية من المصنع تعمل بتردد 1 ميجاهرتز وقد نحتاج أن تغير هذه السرعة ليعمل المُتحكِم دائما بسرعة 16 ميجاهرتز، مثل هذه الإعدادات يجب أن تتم مرة واحدة ولا تتغير في المستقبل.. هنا يأتي دور الفيوزات Fuses.

الفيوزات هي وحدات ذاكرة (بتات Bits) يتم برمجتها بصورة مستديمة، هذا يعني أن محتواها لا يتغير بإعادة تشغيل المُتحكِم أو انقطاع الكهرباء أو حتى برمجة الذاكرة الرئيسية (Flash Memory)، هذه الفيوزات تحتفظ بمجموعة خاصة من الإعدادات المطلوبة وجودها بصورة مستديمة.

تنقسم الفيوزات في مُتحكِمات الـ AVR إلى 3 فيوزات رئيسية و هي:

Low Fuse Byte : يتكون من 8 بتات - كل 1 بت تتحكم في خاصية معينة في الـ AVR

HIGH Fuse Byte: أيضاً 8 بت - كل 1 بت تتحكم في مجموعة خصائص

Extended Fuse Byte: في معظم المُتحكِمات تكون أقل من 5 بتات

جميع البتات في الفيوزات الثلاثة السابقة لها وضعان من البرمجة

- **programmed = 0** الفيوز تم برمجته وتفعيل الخاصية التي يتحكم بها
- **unprogrammed = 1** الفيوز لم يُفعل وتم إلغاء الخاصية التي يتحكم بها

يتم تفعيل أحد الأوضاع بكتابة 0 أو 1 في هذا الفيوز، مع ملاحظة أن صفر لا تعني unprogrammed بل العكس - حيث يمكن أن الفيوز يتم تفعيله programmed عندما تضع بداخله القيمة 0 ويتم إلغاؤه عندما تضع بداخله القيمة 1.

تحذير: بعض الفيوزات إذا تم برمجتها بصورة خاطئة قد تتسبب في تعطيل المُتحكِم أو إلغاء إمكانية برمجته مرة أخرى إلا باستخدام مبرمجات خاصة High voltage programmers، لذا كن على حذر عند اختيار قيم الفيوزات وتأكد أكثر من مرة من جميع الإعدادات قبل برمجتها.

يمكنك التعرف على تفاصيل الفيوزات وكيفة تفعيلها وذلك عبر فتح الـ Datasheet الخاصة بمتحكمات الـ AVR والبحث عن جداول Fuse High Byte , Fuse LOW Byte (موجودة في صفحة 260 لـ Datasheet في حالة المُتحكِم ATmega16).

Fuse High Byte

Table 105. Fuse High Byte

Fuse High Byte	Bit No.	Description	Default Value
OCDEN[4]	7	Enable OCD	1 (unprogrammed, OCD disabled)
JTAGEN[5]	6	Enable JTAG	0 (programmed, JTAG enabled)
SPIEN[1]	5	Enable SPI Serial Program and Data Downloading	0 (programmed, SPI prog. enabled)
CKOPT[2]	4	Oscillator options	1 (unprogrammed)
EESAVE	3	EEPROM memory is preserved through the Chip Erase	1 (unprogrammed, EEPROM not preserved)
BOOTSZ1	2	Select Boot Size (see Table 100 for details)	0 (programmed)[3]
BOOTSZ0	1	Select Boot Size (see Table 100 for details)	0 (programmed)[3]
BOOTRST	0	Select reset vector	1 (unprogrammed)

يوضح الجدول قيمة كل بت في جدول الفيوزات الـ High Byte مع القيمة الافتراضية لكل فيوز (هذه هي القيم التي تأتي مبرمجة مسبقاً من المصنع)، وكما نرى بعض الفيوزات تأتي مفعلة بينما بعض يكون غير مفعل بصورة افتراضية.

قبل أن نبدأ مع الفيوزات سنحتاج أن نتعرف على مفهوم الـ Bootload

البووت-لودر هو برنامج (اختياري) صغير يتم تشغيله قبل البرنامج الرئيسي للقيام ببعض الوظائف، فمثلاً أشهر بووت-لودر هو Arduino bootloader هو السر الذي جعل تكلفة لوحات آردوينو منخفضة جداً. يقوم هذا البووت-لودر بوظيفة بسيطة وهي فتح منفذ السيريال UART ونقل البيانات (أن وجدت) إلى ذاكرة المُتحكِم flash memory هذا يعني أنه يقوم ببرمجة المُتحكِم عبر السيريال بصورة ذاتية self programming دون الحاجة لاستخدام أي

جهاز programmer وبالتالي يوفر كثيراً في تكلفة صناعة اللوحات التطويرية ويعطي المُتحكِم إمكانية البرمجة الذاتية عبر منفذ السيريال بدون تكلفة تذكر. فمثلا لوحة Arduino Nano تباع من المواقع الصينية بنحو 3 دولار فقط مما يجعلها أرخص لوحة تطوير في العالم وتشمل المُتحكِم الدقيق atmega328 مع USB to ttl converter

والآن نعود مرة أخرى للفيوزات:

BOOTRST عند تفعيل هذا الفيوز يبدأ المُتحكِم الدقيق بتشغيل الـ bootloader المُسجِل في الذاكرة أولاً وبعد الانتهاء من تنفيذه يتم الانتقال إلى الـ main function، في أغلب الحالات لا يتم استخدام هذا الفيوز مالم تستخدم bootloader، القيمة الافتراضية = 1 مما يعني أن المُتحكِم سيشغل البرنامج الرئيسي وسيتجاهل البووت-لودر.

BOOTSZ(0,1) يتحكم هذا الفيوز في حجم ومكان تسجيل البووت-لودر في الذاكرة ولديهم جدول كامل من الإعدادات موجودة في صفحة 257 (مجدداً إذا لم تكن تنوي استخدام بووت-لودر فلا تقم بتغير إعدادات هذا الفيوز).

EESAVE يتحكم الفيوز في ما سيحدث لذاكرة الـ EEPROM عند برمجة المُتحكِم، عندما يتم وضع قيمته = 1 سيقوم المُتحكِم بمسح الـ EEPROM في كل مرة يتم رفع ملف هيكس جديد وعندما تكون قيمته = 0 فإن المُتحكِم لن يقوم بمسح محتوى الـ EEPROM أبداً مهما كان عدد مرات البرمجة. القيمة الافتراضية = 1

CKOPT يتحكم في طاقة المذبذب clock oscillator والمكثفات الداخلية (سيتم شرحه بالتفصيل في الجزء الخاص بالمذبذبات).

SPIEN تفعيل خاصية البرمجة عبر منفذ الـ SPI، هذا الفيوز فائق الأهمية حيث يتحكم في تفعيل برمجة الذاكرة. وعندما يتم وضع قيمته = 1 يتم إلغاء إمكانية برمجة المُتحكِم مرة أخرى وعندها لن تستطيع أن ترفع عليه أي برامج جديدة، القيمة الافتراضية = 0 وتعني أن البرمجة عبر الـ spi تعمل. بعض الشركات تستخدم هذا الفيوز لتمنع أي شخص من إعادة برمجة المُتحكِم الدقيق مرة أخرى.

لا تقم بتغيير الفيوز SPIEN أبداً طالماً أنك تتدرب على برمجة المعالج وإلا لن تتمكن من استخدام المُتحكِم مرة أخرى

JTAGEN, OCDEN كلا الفيوزان يستخدمان للتحكم في تفعيل خاصية التنقيح عبر بروتوكول الـ Jtag العالمي.

ما هو الـ Jtag

Jtag بروتوكول عالمي Universal protocol يتوفر في معظم المُتحكِمات المختلفة في العالم سواء كانت PIC, AVR, ARM أو حتى المصفوفات المنطقية FPGA ويستخدم في العديد من الأشياء المفيدة أهمها عملية التنقيح Debugging وقراءة محتويات المُتحكِم الداخلية.

فمثلاً قد نقوم بكتابة برنامج يقرأ قيمة مُسجِل ما ويتخذ قرار معين بناء على هذه القيمة لكن عند تنفيذ البرنامج نجد أن البرنامج يتصرف بصورة خاطئة، هذا التصرف الخاطئ يسمى bug – يعمل الـ Jtag debugger على الوصول إلى محتويات المعالج وإيقافه في لحظة ما وقراءة جميع المحتويات مثل قيمة المتغيرات في الذاكرة أو محتوى المُسجِلات أو الأمر إلى سيتم تنفيذه في الـ Program counter .. الخ.

كما يمكن استخدام الـ Jtag لتشغيل المُتحكِم بخاصية single instruction execution والتي تعني تنفيذ كل أمر بصورة يدوية، يعني أن المُتحكِم لن يقوم بتنفيذ الأوامر البرمجية بصورة متتالية ولكن سينفذ كل أمر على حدى عندما تأمره أنت بذلك.

بهذه المميزات يساعدك الـ Jtag debugger على تتبع الأخطاء واصلاحها والتأكد من أن المُتحكِم يعالج البيانات بصورة صحيحة. كما أن بروتوكول الـ jTag يُستخدَم أيضاً في برمجة المُتحكِمات المختلفة وحتى الـ FPGA.
يمكنك قراءة المزيد عن هذا البروتوكول بصورة مفصلة من الرابط التالي:

en.wikipedia.org/wiki/Joint_Test_Action_Grou

www.ATmel.com/webdoc/atmelice/atmelice.using_ocd_physical_jtag.hl

لاحظ أن ATmega16 يأتي مفعلاً بخيار تشغيل الـ jtag وهذا الخيار يلغي إمكانية استخدام معظم أطراف البورت PORTC كمنافذ IN/OUT لذا، إذا كنت تريد استخدام هذا البورت بالكامل يجب عليك أن تلغي تفعيل الـ jtag بوضع قيمة JTAGEN =1

LOW Byte Fuse

Table 106. Fuse Low Byte

Fuse Low Byte	Bit No.	Description	Default Value
BODLEVEL	7	Brown-out Detector trigger level	1 (unprogrammed)
BODEN	6	Brown-out Detector enable	1 (unprogrammed, BOD disabled)
SUT1	5	Select start-up time	1 (unprogrammed)(1)
SUT0	4	Select start-up time	0 (programmed)(1)
CKSEL3	3	Select Clock source	0 (programmed)(2)
CKSEL2	2	Select Clock source	0 (programmed)(2)
CKSEL1	1	Select Clock source	0 (programmed)(2)
CKSEL0	0	Select Clock source	1 (unprogrammed)(2)

مجموعة الفيوزات الـ **CKSEL(0,1,2.3)** سيتم شرحها بالتفصيل في جزء المذبذبات

قبل شرح الفيوزات BOD يجب أن نتعرف على مفهوم جديد أيضا وهو الـ Brown Out Detection أو ما يعرف اختصاراً بـ BOD.

تعمل هذه الخاصية على إيقاف تشغيل المُتحكِم الدقيق عندما يقل فرق الجهد المطبق عليه عن حد معين. **وهذا سيدفعنا للتساؤل - لماذا قد نحتاج أن نوقف تشغيل المُتحكِم عندما يقل فرق الجهد ؟؟**

العديد من المُتحكِمات الدقيقة (ليس الـ AVR فقط) تتطلب فرق جهد معين ليعمل عند سرعة محددة بصورة مستقرة. فمثلا المُتحكِم ATmega16 يمكنه العمل بسرعة 1 ميجا بفرق جهد 1.8 فولت ولكن إذا تم تشغيله بسرعة 16 ميجاهرتز يجب أن يكون فرق الجهد الذي يعمل به = خمسة فولت على الأقل وحتى 5.5 فولت على الأكثر. وإذا انخفض الجهد عن الخامس فولت سيحدث اضطراب للمذبذب ولن يقوم بتوليد نبضات زمنية صحيحة.

بالتالي فان أي أمر delay أو أي أمر معتمد على عامل زمني سيتم تنفيذه بصورة خاطئة. أيضاً انخفاض الجهد عند تشغيل المُتحكِم على سرعات عالية قد يصحبه أخطاء في قراءة الذاكرة والـ EEPROM وقد يتسبب في أخطاء أخرى عشوائية. لذا يتم استخدام الـ BOD لإيقاف تشغيل المُتحكِم إذا كان فارق الجهد المطبق عليه أقل من المطلوب ويتم إعادة تشغيله تلقائياً عندما يعود فارق الجهد للمستوى المناسب ليعمل المُتحكِم بكفاءة.

الحقيقة أنه لا داعي لتشغيل الـ BOD إلا إذا كنت ستشغل المُتحكِم بسرعة أكبر من 4 ميجاهرتز لأن معظم مُتحكِمات الـ AVR يمكنها العمل بصورة طبيعية حتى فرق جهد 1.8 فولت على سرعات من 1 إلى 4 ميجا.

BODEN يقوم هذا الفيوز بتفعيل خاصية الـ BOD ويأتي بقيمة 1 (غير مفعل) بصورة افتراضية

BODLEVEL يتحكم الفيوز في فارق الجهد الذي سيتم إيقاف المُتحكِم عن التشغيل عندما يصل إليه، فإذا كانت قيمة الفيوز = 1 هذا يعني أن المُتحكِم سيتم إيقاف تشغيله عند فرق جهد أقل من 2.7 فولت أما إذا كانت قيمة الفيوز = 0 هذا يعني أن المُتحكِم سيتوقف عند فرق جهد أقل من 4 فولت (مع ملاحظة أن هذه الإعدادات تصبح فعالة إذا كانت قيمة الـ BODEN مفعلة أي أنها تساوي صفر).

SUT(0,1) تتحكم هذه الفيوزات في الزمن الذي سيستغرقه المُتحكِم من بداية توصيل الكهرباء حتى البدء في تنفيذ البرنامج المُسجِل بداخله أو الوقت المستغرق لعمل Reset. يمكنك ضبط هذا الفيوز بمجموعة من القيم المختلفة (الجدول يوضح زمن تأخير التشغيل عند استخدام المذبذب الداخلي مع ملاحظة أن هذه الإعدادات تختلف عند استخدام كريستالة خارجية أو مذبذب من نوع أخر).

Table 10. Start-up Times for the Internal Calibrated RC Oscillator Clock Selection

SUT1..0	Start-up Time from Power-down and Power-save	Additional Delay from Reset (V_{CC} = 5.0V)	Recommended Usage
00	6 CK	–	BOD enabled
01	6 CK	4.1 ms	Fast rising power
10(1)	6 CK	65 ms	Slowly rising power
11	Reserved		

كما هو موضح في الجدول، نجد أن تغيير الإعدادات سيجعل المُتحكِم يتأخر إما 0 أو 4 أو 65 مللي ثانية عند بداية التشغيل (يعني أنه سيستغرق 64 مللي ثانية من بدء توصيل الكهرباء إلى أن يبدأ تنفيذ البرنامج).

لماذا نحتاج أن نجعل المُتحكِم يتأخر فترة ما قبل أن يبدأ تنفيذ البرنامج؟

هناك سببين لهذا الأمر.

- **الأول**: بعض الدوائر والعناصر الإلكترونية التي قد تتصل بالمتحكم الدقيق تحتاج القليل من الوقت بعد توصيل الكهرباء حتى تكون جاهزة للعمل بصورة مستقرة لذا يستحسن أن يتأخر المُتحكِم الدقيق قليلاً قبل أن يبدأ العمل أو الاتصال مع هذه المكونات.

- **الثاني**: معظم دوائر المذبذبات Oscillator التي تولد الـ CPU Clock تحتاج لوقت كبير **نسبياً** حتى تستقر النبضات المتولدة منها (مثل دوائر الـ RC oscillator وبعض الـ ceramic oscillator) لذا يجب على المُتحكِم أن يتأخر قليلاً حتى تستقر ذبذبات الساعة الداخلية.

يستحسن أن تجعل التأخير الزمني = 65 مللي ثانية لضمان أفضل أداء للمُتحكِم وهذا يعني أن تجعل قيمة الـ SUT1 – SUT0 كلاهما = 1

LockBits 6.2

مثل الفيوزات تتحكم الـ lockbits في مجموعة من الخصائص الثابتة التي لا تتغير إذا تم برمجتها، هذه الخصائص هي "إمكانية الوصول وحماية الذاكرة". حيث يمكنك استخدام الـ lockbits في حماية البيانات على ذاكرة المُتحكِم الرئيسية flash من النسخ أو القراءة كما يمكنك حماية بعض أجزاء الذاكرة eeprom من الكتابة عليها ومنع أي برنامج أو شخص من الوصول إلى محتواها.

أيضاً تستخدم الـ lockbits في تخصيص جزء ثابت من الذاكرة لا يقبل التعديل بعد برمجته مثل ما يحدث مع الـ bootloader (أشهر مثال للبوتلودر هو arduino bootloader) وهو البرنامج المسؤول عن استقبال ملف الـ hex من السيريال بدل من الـ SPI.

كما نرى في الجدول التالي اعدادات الـ lockbit التي يمكنها أن تغلق إما القراءة والكتابة أو القراءة فقط من جميع أنواع الذاكرة Flash & EEPROM وذلك عبر التحكم في قيم الـ LB1 و LB2

Table 104. Lock Bit Protection Modes

Memory Lock Bits[2]			Protection Type
LB Mode	**LB2**	**LB1**	
1	1	1	No memory lock features enabled.
2	1	0	Further programming of the Flash and EEPROM is disabled in Parallel and SPI/JTAG Serial Programming mode. The Fuse bits are locked in both Serial and Parallel Programming mode.[1]
3	0	0	Further programming and verification of the Flash and EEPROM is disabled in Parallel and SPI/JTAG Serial Programming mode. The Fuse bits are locked in both Serial and Parallel Programming mode.[1]

- عندما تكون LB1 = 1 و LB2 =1 لا يتم تفعيل الـ lockbit **ويسمح بالقراء والكتابة من وإلى الذاكرة**
- عندما تكون LB1 = 0 و LB2 =1 يتم تفعيل الحماية و**يمنع أي محاولة (كتابة) بيانات** على كل من الـ Flash والـ EEPROM
- عندما تكون LB1 = 0 و LB2 =1 يتم تفعيل الحماية و**يمنع أي محاولة (كتابة) أو (قراءة** البيانات على كل من الـ Flash والـ EEPROM

6.3 المذبذبات والـ Clock Source



شكل نبضات الساعة عبر الزمن

عندما نشتري أجهزة الحاسوب الخاصة نحاول دائما شراء الجهاز الأسرع وهنا نجد الشركات المصنعة دائماً تتباهي بأن منتجاتها هي الأفضل لأنها تعمل بأعلى سرعة معالجة (تقاس بالجيجاهرتز - مليار هرتز). ومثل أجهزتنا الشخصية نجد أن سرعة المُتحكِمات الدقيقة هي أيضاً تحدد عبر التردد الذي تعمل عليه أو كما تسمى (مصدر الساعة clock source).
حيث يتم تنفيذ الأوامر بسرعة = 1÷(التردد) الذي يعمل به المُتحكِم، على سبيل المثال. جميع شرائح الـ AVR تقوم بتنفيذ أمر واحد كل 1 نبضة من الـ clock source.

- إذا كان التردد = 1 ميجاهرتز **(السرعة الافتراضية لمعظم مُتحكِمات الـ AVR)** هذا يعني أن المُتحكِم يمكن تنفيذ 1 مليون أمر بلغة الأسمبيلي في الثانية الواحدة ويستغرق الأمر 1 ميكروثانية
- إذا كان التردد = 1 ميجاهرتز هذا يعني أن المُتحكِم يمكنه تنفيذ 1 مليون أمر بلغة الأسمبيلي في الثانية الواحدة ويستغرق الأمر 1 ميكروثانية
- إذا كان التردد = 2 ميجاهرتز هذا يعني أن المُتحكِم يمكنه تنفيذ 2 مليون أمر بلغة الأسمبيلي في الثانية الواحدة ويستغرق الأمر نصف ميكروثانية أو (500 نانوثانية).
- إذا كان التردد = 4 ميجاهرتز هذا يعني أن المُتحكِم يمكنه تنفيذ 4 مليون أمر بلغة الأسمبيلي في الثانية الواحدة ويستغرق الأمر 250 نانوثانية nano second
- إذا كان التردد = 8 ميجاهرتز هذا يعني أن المُتحكِم يمكنه تنفيذ 8 مليون أمر بلغة الأسمبيلي في الثانية الواحدة ويستغرق الأمر 125 نانو ثانية

النانو ثانية Nano Second (nS) = جزء من مليار جزء من الثانية
الميكرو ثانية Micro Second (uS) = جزء من مليون جزء من الثانية.

تدعم مُتحكِمات الـ AVR العديد من تقنيات توليد الذبذبات اللازمة لتشغيل المُتحكِم أو كما تمسى Clock Source تختلف هذه التقنيات في مدى دقتها ونسبة الخطأ والتكلفة المالية، في هذه الجزء سنستعرض 4 طرق مختلفة مع إيضاح مميزات وعيوب كل طريقة.

Internal Calibrated RC oscillator

كلمة RC oscillator تعني "المذبذب الداخلي المصنوع بمقاومة ومكثف" وتعتبر هذه الطريقة هي أرخص أسلوب للحصول على الـ Clock source حيث تحتوي شرائح الـ AVR ومنها ATmega16 على مذبذب RC داخلي يمكنه العمل بسرعات من 1 إلى 8 ميجاهرتز (يتم ضبطها بالفيوزات كما سنرى لاحقاً). ولا تتطلب هذه الطريقة توصيل أي مكونات إضافية وبالتالي يتم تصغير حجم المشروع وتقليل التكلفة

العيوب: نسبة الخطأ في التردد والتوقيت 3% - هذه النسبة قد لا تؤثر في معظم التطبيقات، لكنها خطيرة جداً في التطبيقات التي تحتاج توقيت دقيق أو تستخدم دوال delay_ms على فترات بعيدة. **على سبيل المثال**: إذا كان مشروعك عبارة عن تشغيل load كل ثانية واحدة فلن تشعر بهذا الخطأ لأن التوقيت الحقيقي = 1 ثانية (+\-) نسبة الخطأ = (0.03 * 1 = 0.03) وهو ما يساوي 1.03 ثانية أو 1 ثانية طرح 0.03 = 0.97 ثانية.

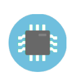
لكن تخيل أن مشروعك يقوم بتشغيل الـ load كل 1 ساعة = 3600 ثانية، عندها ستجد أن الوقت الحقيقي يصبح 3600 +\- (0.03 * 3600) وهو ما يساوي 3708 ثانية (هذا يعني أنه هناك (تأخير أو تقديم) 108 ثانية إضافية والذي يعتبر رقم كارثي وبعيد تماماً عن التوقيت المطلوب).
متى نستخدمها؟ إذا كان مشروعك لن يتواجد به أي تأخير زمني يزيد عن بضعة ثواني فقط (أقل من 10 ثواني) وتريد أن تصغر تكلفة المشروع.

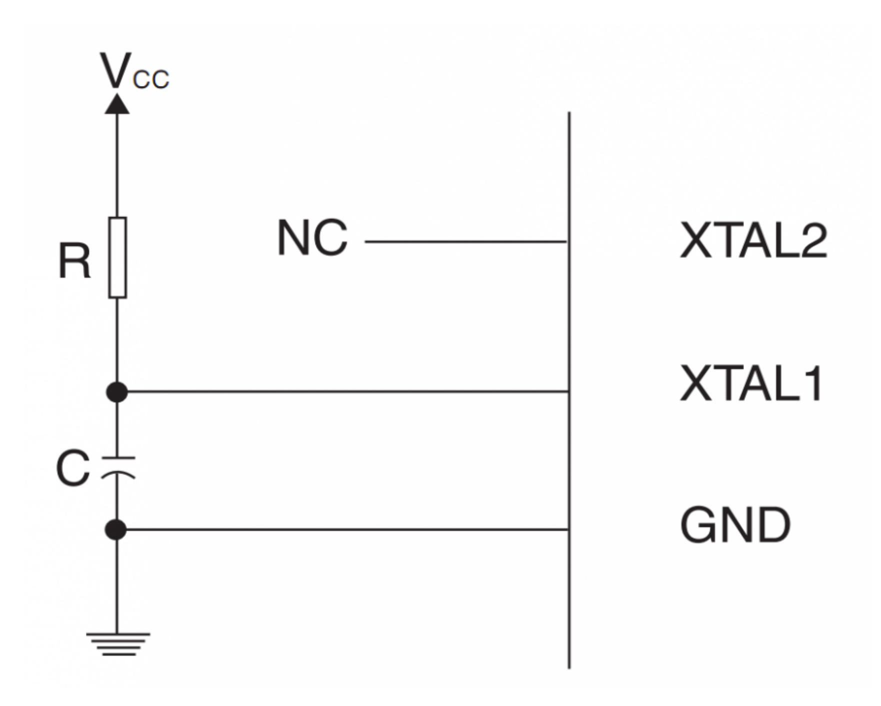
External RC Circuit

هذه الطريقة مماثلة للسابقة ولكن الاختلاف الوحيد أن دائرة الـ RC تكون موجودة خارج شريحة الـ AVR وتوفر ترددات أكبر من 8 ميجا حيث يمكنك ضبط التردد عبر التحكم في قيمة المقاومة والمكثف عبر القانون التالي:

$$F(frequency)=\frac{1}{(R*C)}$$

حيث تمثل R قيمة المقاومة بالأوم، و C قيمة المكثف بالفاراد ويتم توصيل الدائرة على الطرف XTAL1 بالطريقة التالية (NC يعني غير متصل بشيء)





المميزات: يمكنك الحصول على ترددات أكبر من الـ internal oscillator بتكلفة قليلة كما يمكنك تغير التردد في أي وقت عبر وضع **مقاومة متغيرة** بدلاً من المقاومة الثابتة.

العيوب: مثل النوع السابق، غالباً لا يمكنك تحديد قيمة المكثف ولا المقاومة بدقة كبيرة حيث تحتوي هذه العناصر على نسبة خطأ 5% وبالتالي التردد الحقيقي الناتج عنها يكون مضاف إليه نسبة خطأ 5%

في حالة أنك تريد استخدام هذه الطريقة يجب الانتباه إلى قيمة المكثف حيث يجب أن تكون 22 بيكوفاراد أو أكثر ويجب أن يكون من النوع السيراميكي (منعدم القطبية) ceramic capacitor

> معلومة إضافية: بعض مُتحكِمات AVR تحتوي على مكثف داخلي بقيمة 36 بيكوفاراد وبالتالي لن تحتاج سوى أن توصل مقاومة فحسب.

External Crystal “Quartz” Oscillator

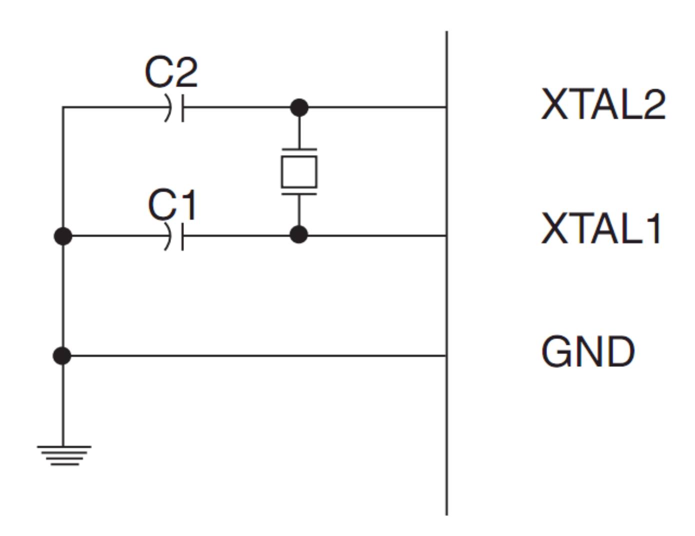
تعتبر هذه الطريقة هي الأكثر شيوعاً والمفضلة لدى جميع الشركات ومصممي الأنظمة المدمجة، حيث يتم استخدام الكريستالة ذات الطرفين لتوليد التردد المطلوب بدقة عالية جداً وبأقل نسبة خطأ ممكنه حيث تبلغ نسبة الخطأ في الكريستالة حوالي 10 هرتز من أصل 1 مليون وهو ما يساوي 0.00001 واحد من كل مئة آلف وهذا يعني أنها أكثر بدقة بنحو 1000 مرة من الـ RC oscillator.

المميزات: الدقة العالية جداً وثبات التردد مهما تغيرت درجات الحرارة وبالتالي فهي توفر أداء ممتاز طوال فترة التشغيل

العيوب: التكلفة حيث تحتاج هذه الكريستالات إلى مكثفات إضافية عدد 2 مكثف بسعة 22 بيكوفاراد مما يزيد التكلفة لتصل إلى 1 دولار تقريباً (تكلفة الكريستالة بمفردها = نصف دولار).

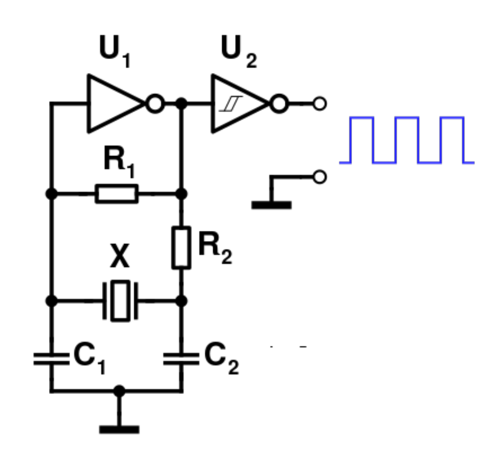
أشهر هذه الكريستالات هي 16 ميجا والـ 12 ميجا والـ 8 ميجا والـ 24 ميجا، في هذا الكتاب سنستخدم الـ 16 ميجا في بعض التجارب (وهو أقصى تردد يمكن لشريحة الـ ATmega16 أن تعمل به مع العلم أن المتحكم ATTiny يمكنه العمل بسرعة تصل إلى 20 ميجاهرتز).

يتم توصيل الكريستالة والمكثفات بالصورة التالية:

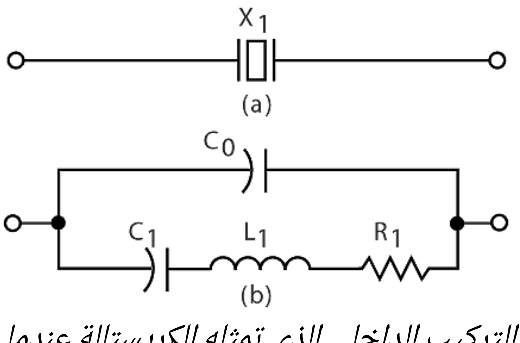


معلومة إضافية: الكريستالة هي عنصر إلكتروني مصنوع من مادة بلورية اسمها "الكوارتز Quartz" وعندما يتم توصيلها بالكهرباء فإنها تعمل **كأنها** دائرة رنين مكونة من مكثف وملف ومقاومة R-L-C circuit تنتج Sin wave ولكن عندما تتصل بدائرة pierce oscillator داخل المُتحكِم الدقيق يتم تحويل الذبذبات المتولدة على هيئة sin wave إلى pulse باستخدام عاكس inverter يمكنك معرفة المزيد عن الـ pierce oscillator من الرابط التالي:
http://www.abracon.com/Support/facn_abracon_jul2011.pdf

ويمكنك قراءة صفحة الكريستالات على ويكيبيديا لتتعرف أكثر عليها.
en.wikipedia.org/wiki/Crystal_oscillator



تركيب دائرة المذبذب (الكريستالة + المكثفات + العاكس)



التركيب الداخلي الذي تمثله الكريستالة عندما تتصل بدائرة كهربية

External Resonators

هذا النوع من المذبذبات يشبه لحد كبير الكريستالة وفي الحقيقة هو يوازي "كريستالة + المكثفات الإضافية" في قطعة إلكترونية واحدة ويتوفر في الأسواق بترددات مختلفة بدأ من 1 ميجاهرتز إلى 24 ميجاهرتز







CERAMIC RESONATOR

المميزات: السعر المنخفض (أرخص من الكريستالة) كما أنه يحتوي على المكثفات المطلوبة بداخله وحجمه الصغير وتوافر معظم الترددات المطلوبة

العيوب: يتملك نسبة خطأ 0.5% (نصف بالمئة)، وهي تعتبر نسبة أفضل بكثير من الـ internal RC oscillator ولكنها لا تقارن بدقة الكريستالة الكوارتز.

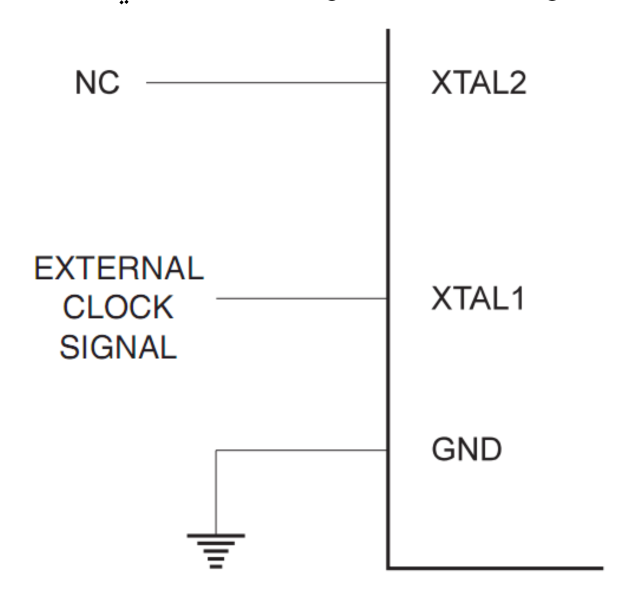
External Pure Pulse (TTL) Oscillator



المذبذبات الأكثر دقة على الإطلاق حيث تبلغ نسبة الخطأ 5 هرتز لكل 1 مليون هرتز وهو ما يساوي 0.000005 والذي يعني أنها تمتلك دقة توازي 20 ضعف دقة الكريستالات الكوارتز.

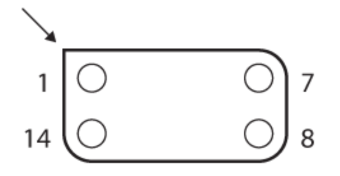
تتوفر هذه المذبذبات بأشكال عديدة أغلبها تكون مربعة أو مستطيله الشكل وتمتلك 4 أطراف VCC – CLK – GND والطرف الرابع غير مستخدم. يتم توصيل الطرف CLK بالطرف XTAL1 في شريحة المُتحكِم AVR وتوصيل الطرف VCC بنفس مصدر الجهد الخاص بالـ AVR (سواء كان 5 فولت أو 3.3 فولت).

العيوب: السعر المرتفع، حيث يبلغ سعر هذا النوع من المذبذبات نحو 3 دولار أو أكثر (يزداد السعر بزيادة التردد المطلوب) وفي بعض الأحيان يكون سعرها أكبر من سعر المُتحكِم الدقيق نفسه. الصور التالية توضح توصيل الـ AVR مع المذبذب الخارجي TTL oscillator



الصور التالية توضح أشكال وأحجام مختلفة للمذبذب





Pin	Function
1	NC
7	GND
8	Output
14	+5VDC

6.4 قيم الفيوزات لضبط السرعة

القيم التالية هي المستخدمة في التحكم بسرعة شريحة ATmega16 – جميع القيم تم أخذها من ملف الـ DataSheet الرسمي للمُتحكِم (والمرفق مع الكتاب) بداية من الصفحة رقم 24 مع ملاحظة أن هذه الفيوزات تعمل على كل من ATmega16/ ATmega32 لأنهم من نفس الفئة من المُتحكِمات (وقد تختلف هذه الفيوزات في فئات أخرى من شرائح الـ AVR). - جميع القيم مخصصة للفيوزات **Clock Select** أو كما تعرف اختصاراً بي **CKSEL**

فيوزات ضبط السرعة مع المذبذب الداخلي Internal RC

قم بضبط الفيوزات بالقيم التالية لاختيار السرعة المطلوبة، مع ملاحظة أن القيمة هي ,, CKSEL3, CKSEL2, CKSEL1, CKSEL0 بنفس الترتيب حيث يعبر كل 1 بت عن قيمة الـ CKSEL الموازية له.

CKSEL(3,2,1,0)	Clock Frequency
0001	1 Mhz
0010	2 Mhz
0011	4 Mhz
0100	8 Mhz

فيوزات ضبط السرعة مع دائرة External RC

CKSEL(3,2,1,0)	Clock Frequency
0101	التردد أقل من 0.9 ميجاهرتز
0110	التردد المتوقع أكبر من 0.9 وأقل من 3 ميجاهرتز
0111	التردد المتوقع أكبر من 3 وأقل من 8 ميجاهرتز
1000	التردد المتوقع أكبر من 8 وأقل من 12 ميجاهرتز

فيوزات ضبط السرعة مع External Crystal or Ceramic

هذه الإعدادات يمكن استخدامها مع كل من الـ Crystal oscillator والـ ceramic resonator مع ملاحظة أنه في حالة استخدام الكريستالة يجب أن يتم وضع مكثفات إضافية معها كما ذكرنا سابقاً ولا يتم وضع هذه المكثفات مع الـ ceramic resonator.

أيضاً لاحظ أن الإعدادات يتم وضعها للفيوزات **CKSEL** من 1 إلى 3 فقط ولا يتم برمجة الـ CKSEL0 معهم في هذا الوضع. ويتم برمجة الفيوز **CKOPT** في حالة استخدام كريستالة أكبر من 8 ميجاهرتز.

CKSEL(3,2,1) **"0 is not used"**	**Clock Frequency**	**سعة المكثفات المقترحة**
101 CKOPT = 1	هذا الوضع يُستخدَم مع ceramic resonator وفي حالة أن التردد المطلوب ما بين 0.4 إلى 0.9 ميجاهرتز	لا يتم استخدام مكثفات
110 CKOPT = 1	التردد المتوقع أكبر من 0.9 وأقل من 3 ميجاهرتز	السعة المقترحة بين 12 → 22 picofarad
111 CKOPT = 1	التردد المتوقع أكبر من 3 وأقل من 8 ميجاهرتز	السعة المقترحة بين 12 → 22 picofarad
111 CKOPT = 0	التردد المتوقع أكبر من 8 ميجا وحتى 16 ميجاهرتز	السعة المقترحة بين 12 → 22 picofarad

ما هي أهمية الفيوز CKOPT؟

هذا الفيوز يتحكم في بعض الأمور المهمة، منها تشغيل المذبذب بالطاقة القصوى أو الطاقة المنخفضة (وضع توفير الطاقة عندما يكون CKOPT = 1 “unprogrammed”fuse) هذا الوضع يساهم في تخفيض الطاقة التي يستهلكها المُتحكِم الدقيق لكنه لا يصلح لتشغيل الكريستالات الأكبر من 8 ميجاهرتز لأنه كلما زاد تردد الكريستالة كانت النبضات الناتجة منها أضعف من ناحية فرق الجهد وبالتالي قد لا تصلح لتشغيل المعالج.

للتغلب على هذه المشكلة يتم تفعيل الـ CKOPT والذي سيقوم بتشغيل المذبذب بالطاقة القصوى أو كما يسمى Full rail-to-rail swing مما يجعل المعالج يحصل على أفضل نبضات ممكنه تكفي لتشغيله في السرعات العالية وتكفي أيضاً لإخراج نبضات دقيقة لتشغيل المكونات الخارجية.

من المفيد جداً تفعيل هذا الفيوز في الدوائر التي ستتعرض إلى noise أو ستوضع في مكان معرض لإشعاع كهرومغناطيسي كبير نسبياً حيث يساعد وضع Full rail-to-rail على تحسين أداء المُتحكِم في البيئات ذات الـ noise الكبيرة.

الوظيفة الثانية له هي تفعيل المكثف الداخلي "سعة 36 بيكوفاراد" لتشغيل دائرة الـ external RC circuit بالمقاومة فقط دون مكثف إضافي.

فيوزات ضبط السرعة مع المذبذب الخارجي Pulse Oscillator

عند استخدام أي مصدر خارجي لنبضات الساعة pulses مثل الـ TTL oscillator أو حتى شريحة IC 555 يتم وضع كل قيم **CKSEL(3,2,1,0) = 0000**

ملخص إعدادات الفيوزات للتحكم بالسرعة

- إذا أردت تشغيل المُتحكِم الدقيق بأقل استهلاك ممكن للطاقة قم بتفعيل التردد على 1 ميجاهرتز فقط عبر وضع CKSEL(3,2,1,0) = 0001.
- إذا أردت تشغيل المُتحكِم بأقصى سرعة (8 ميجاهرتز) دون استخدام أي مكونات إضافية قم بوضع القيم CKSEL(3,2,1,0) = 0100.
- إذا كنت تنوي أن تستخدم أي كريستالة خارجية فالأفضل أن تشغل الـ Full rail-to-rail عبر تفعيل الفيوز CKOPT بوضع القيمة 0 بداخله واختيار CKSEL(3,2,1) = 111 – هذا الوضع سيعمل بصورة ممتازة مع جميع الكريستالات.

ملاحظات هامة بخصوص تعديل السرعة

يجب الانتباه عند تغير سرعة المُتحكِم الدقيق حيث أن سرعة رفع البرنامج (ملف الهيكس) يجب أن تكون أقل من 1/8 من تردد الـ Clock المستخدم.

علي سبيل المثال إذا كانت سرعة المُتحكِم = 1 ميجاهرتز إذا يجب أن تكون سرعة رفع البرنامج على المُتحكِم أقل من 128 كيلوهرتز وإلا قد تجد خطأ من برنامج AVRdude مفاده أن البرنامج لا يستطيع أن يتواصل مع المُتحكِم الدقيق.

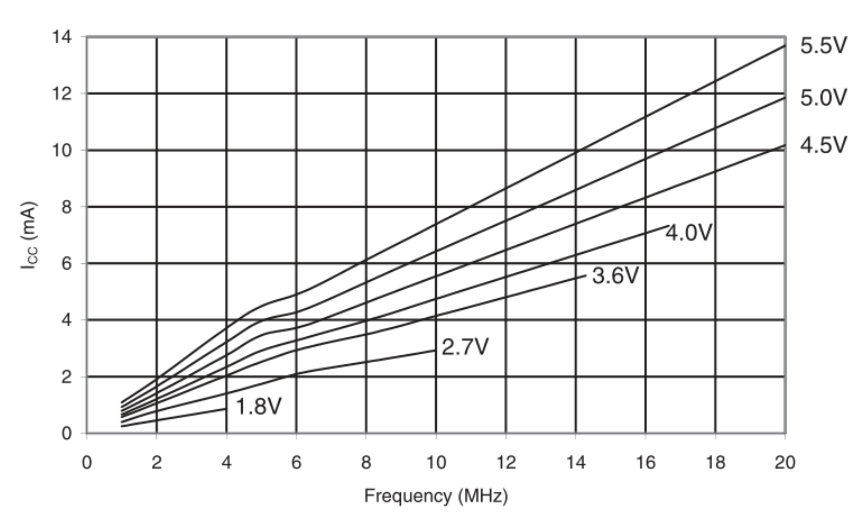
على أي حال إذا قرأت هذا الخطأ فكل ما عليك فعله هو تقليل سرعة الرفع عن طريق توصيل طرفي الـ jumper الموجودة بدائرة المُبرمجة USBasp (ستجد كلمة slow مكتوبة بجانبها)وإذا كنت تستخدم مبرمجة أخرى لا تحتوى هذا الوضع فيمكنك أن تختار سرعة الرفع من برنامج AVRdudess (اجعل قيمتها = 500 هرتز فقط).

عند استخدام الدالة _delay_us(time) يجب أن يعمل المذبذب بكريستالة خارجية بتردد 8 ميجاهرتز على الأقل حتى تعمل الدالة بتوقيت صحيح (إذا تم استخدام المذبذب الداخلي أو كريستالة بتردد أقل فيحدث خطأ في الدالة delay_us ولن يتم التأخير بالوقت المطلوب)

6.5 الطاقة وسرعة تشغيل المُتحكِمات

في العديد من التطبيقات يجب أن يتم تصميم النظام المدمج بحيث يعمل على مصدر منخفض جداً للطاقة مثل **بطارية** أو **لوحة شمسية** صغيرة، من أجل هذه التطبيقات يتم ضبط المُتحكِم للعمل على الترددات المنخفضة.

معظم المُتحكِمات الدقيقة (وبالتحديد المعالجات الدقيقة الموجود بداخلها) **تستهلك طاقة أكثر كلما كانت تعمل بتردد أكبر** وبالتالي نجد أن تخفيض التردد يقلل بدرجة كبيرة جداً استهلاك التيار الكهربي كما هو ملحوظ في الرسم البياني التالي:



معدلات استهلاك الطاقة لعائلة ATTiny وبعض أفراد عائلة atmega

كما نرى على المحور الرئيسي (استهلاك التيار بالمللي أمبير) والمحور الأفقي التردد (بالميجاهرتز) وكل خط مرسوم يمثل فرق الجهد الذي يعمل عنده المُتحكِم الدقيق. من هذا الرسم نستنتج العديد من الأمور.

- عند فرق جهد 1.8 ميجا يمكن للمُتحكِم أن يعمل بتردد من 1 ميجا إلى 4 ميجا بحد أقصى وباستهلاك تيار من الثاني حتى 1.8 مللي أمبير فقط.

- عند فرق جهد 2.7 فولت يمكن للمُتحكِم أن يعمل بسرعة تصل إلى 10 ميجاهرتز واستهلاك تيار حوالي 3 مللي أمبير.
- عند فرق جهد 3.6 يمكن للمُتحكِم أن يعمل بتردد يصل إلى 14.5 ميجاهرتز تقريباً وباستهلاك تيار يصل إلى 5.7 مللي أمبير.
- عند فرق جهد 4 فولت يمكن للمُتحكِم أن يعمل حتى 17 ميجاهرتز باستهلاك تيار 7 مللي أمبير.
- عند فرق جهد من الخامس إلى 5.5 يمكن للمُتحكِم أن يعمل بسرعة تصل إلى 20 ميجاهرتز.

ملاحظة: الحد الأقصى لتردد الـ ATmega16/ATmega32 هو 16 ميجاهرتز فقط، بينما المُتحكِمات الأحدث منها مثل ATmega328 أو عائلة ATTiny يمكنها أن تصل إلى 20 ميجاهرتز

بعض مُتحكِمات الـ AVR تدعم العمل بتردد 125 كليوهرتز أو أقل، هذا المعدل يجعل استهلاك التيار الكهربي منخفض جداً لدرجة أنه يصل إلى 100 ميكرو أمبير (100 جزء من المليون من الأمبير وهو ما يساوي 0.1 مللي أمبير).

تختلف هذه المقاييس قليلاً بتغير درجة الحرارة التي يعمل عندها المعالج وتوجد رسومات بيانية مفصلة تشرح معدلات استهلاك الطاقة بالتفصيل عند درجات الحرارة المختلفة بدءاً من صفحة 299 في الـ Datasheet المرفقة (مع ملاحظة أن بعض معدلات استهلاك الطاقة قد تختلف قليلاً مع مُتحكِمات الـ AVR الأخرى).

كيف تحسب عمر البطارية

عند استخدام البطاريات سيكون من المفيد جداً أن تحسب وقت التشغيل حتى تنفذ طاقة البطارية وقد يحدد هذا الوقت التردد المطلوب لتشغيل المُتحكِم. وقبل أن نبدأ الحسابات علينا أن نتعرف على بعض الأمور.

استهلاك الطاقة المذكور مسبقاً هو للمُتحكِم الدقيق نفسه وليس لأي حمل load متصل به ويمكن تخفيض هذا الاستهلاك قليلاً بإيقاف تشغيل الـ ADC

استهلاك الطاقة الكلي Total load = فرق الجهد * (استهلاك التيار للمُتحكِم + الأحمال المتصلة به) + الطاقة الضائعة من الـ voltage regulator إن وجد.

قانون حساب وقت البطارية

$$Battery\ Working\ Time = \frac{0.8 \times Battery\ Capacity\,(mAH)}{Total\ Load\ Current\,(mA)}$$

جميع البطاريات يكون لها سعة تقاس بالمللي أمبير\ساعة فمثلا بطارية الهاتف المحمول نجد مكتوب عليها 3.7 فولت 3000 mAH (مللي أمبير\ساعة) أو تكتب 1 A/hour (لأن كل 1000 مللي أمبير = 1 أمبير). وهذا يعني أنه في حالة تشغيل هذه البطارية على أحمال تستهلك 3000 مللي أمبير فإن البطارية ستظل تعمل 1 ساعة فقط.

مثال: إذا كان المُتحكِم الدقيق يعمل بفرق جهد 3.7 فولت و بتردد 16 ميجاهرتز (يستهلك 7 مللي أمبير) ومتصل به دايود ضوئي يستهلك 7 مللي أمبير كما أن منظم الجهد يستهلك تيار إضافي 6 مللي أمبير. احسب زمن التشغيل على بطارية سعتها 1000 مللي أمبير بافتراض أن جميع الأحمال تعمل بصورة مستمرة دون أن تنطفئ.

الحل

أولاً: نحسب استهلاك التيار الكلي = 7 مللي (للمُتحكِم) + 7 مللي (للدايود) + 6 مللي لمنظم الجهد = 20 مللي أمبير إجمالي استهلاك طاقة.

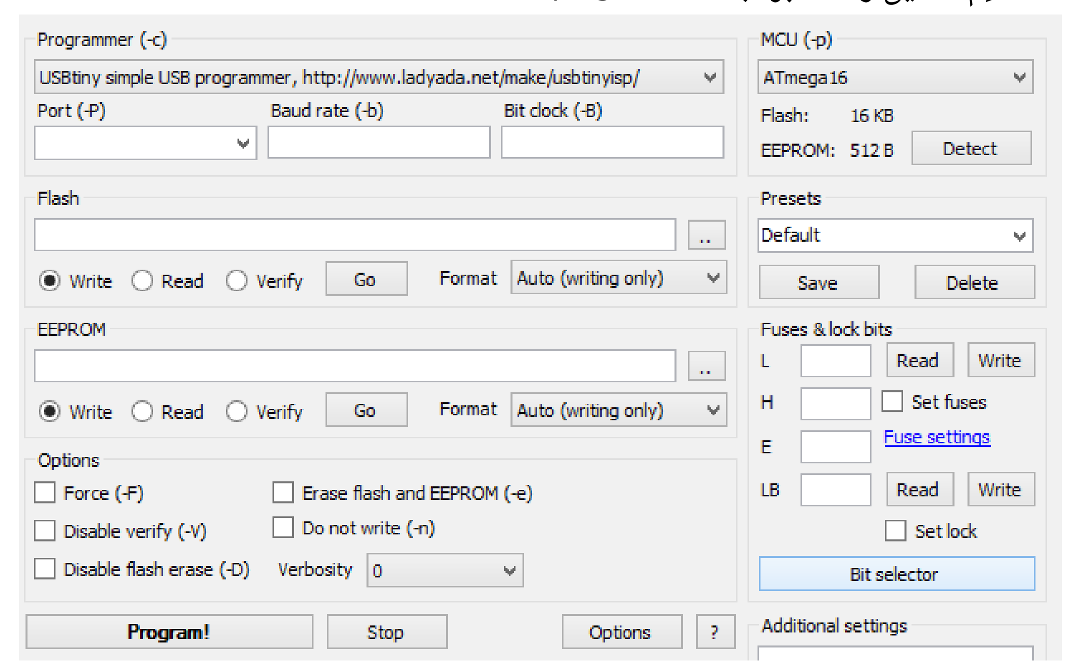
ثانياً: بالتعويض بالقيم في القانون السابق نجد أن ساعات التشغيل = 40 ساعة

Working Time = 0.8 * (1000/20) = 40 hours

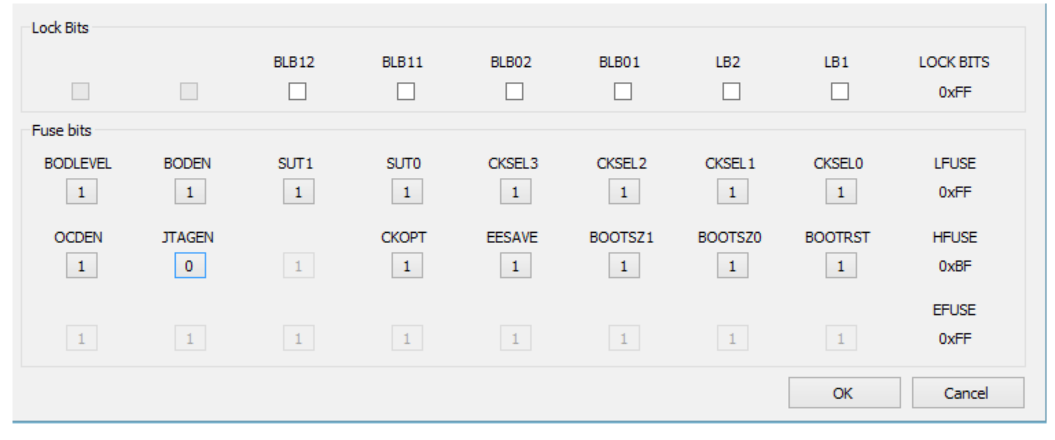
معلومة إضافية: لماذا نضرب الرقم 0.8 في القانون السابق بالرغم من أنه يفترض أن نقسم سعة البطارية على الاستهلاك مباشرة؟ السبب هو تواجد استهلاك ضائع من الطاقة يحدث من **البطارية نفسها بسبب المقاومة الداخلية Battery Internal Resistor** كما أن الأسلاك والتوصيلات في الدائرة الكهربية أيضاً تساهم في ضياع بعض الطاقة خاصة إذا كان هناك أسلاك من معدن الألومنيوم

6.6 كيف تبرمج الفيوزات

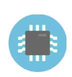
برمجة الفيوزات عملية سهلة للغاية عبر برنامج AVRdudess كل ما عليك فعله هو اختيار نوع المُتحكِم الدقيق وأداة البرمجة programmer

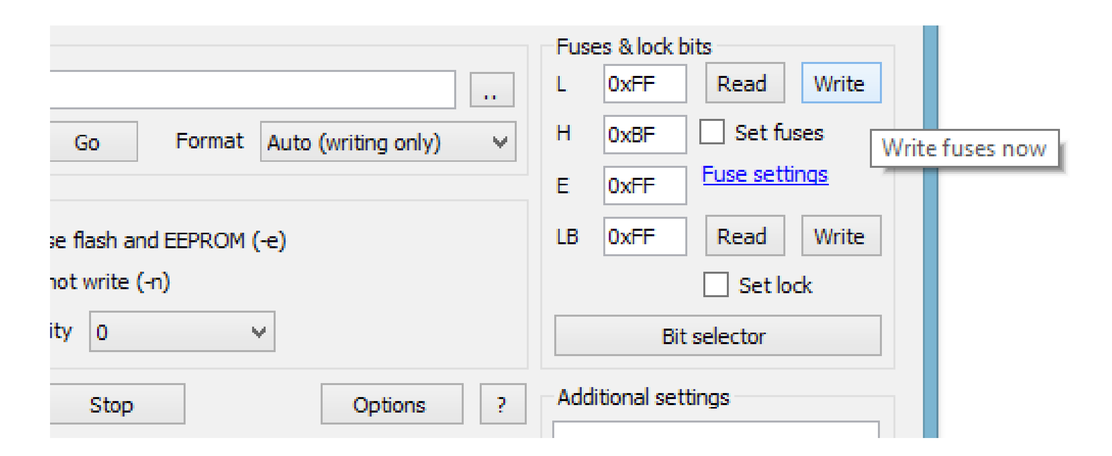


ثم الضغط على زر Bit selector من مربع Fuses & lockbits لتظهر لوحة اختيار قيم الفيوزات (لاحظ أن ATmega16 لا يمتلك Extended Fuses):



اختر تفعيل (أو إلغاء تفعيل) الفيوزات المرغوبة كما تريد ثم اضغط Ok لتجد أن قيمة الـ Low Byte Fuse والـ HIGH Byte Fuse أصبحت جاهزة للبرمجة كما في الصورة التالية





الآن يمكنك الضغط على زر Write الموجود في الأعلى (بجانب الفيوزات) وسيقوم الـ programmer بحرق قيم الفيوزات المطلوبة.

يوفر برنامج AVRdudess خيار Set fuses والذي يعني أن البرنامج سيقوم بإعادة كتابة الفيوزات في كل مرة يقوم فيها برفع ملف هيكس جديد على المُتحكِم الدقيق. **من الأفضل عدم استخدام هذا الخيار والاكتفاء ببرمجة الفيوزات عبر زر Write فحسب.**

أيضا الزر Read بجانب Write والذي عند الضغط عليه سيقوم الـ Programmer بقراءة قيمة الفيوزات على الشريحة المتصله به.

أيضاً ستجد كلمة Fuse Setting وهي عبارة عن رابط لموقع خاص يعمل كآلة حاسبة للفيوزات لكل أنواع شرائح الـ AVR كل ما عليك هو الدخول إليه وتحديد الخصائص التي تريدها وسيخبرك الموقع بأي الفيوزات ينبغي لك أن تفعلها وأي منها لا يجب أن تفعلها.

6.7 كيف تعالج الفيوزات المُبرمجة بصورة خاطئة؟

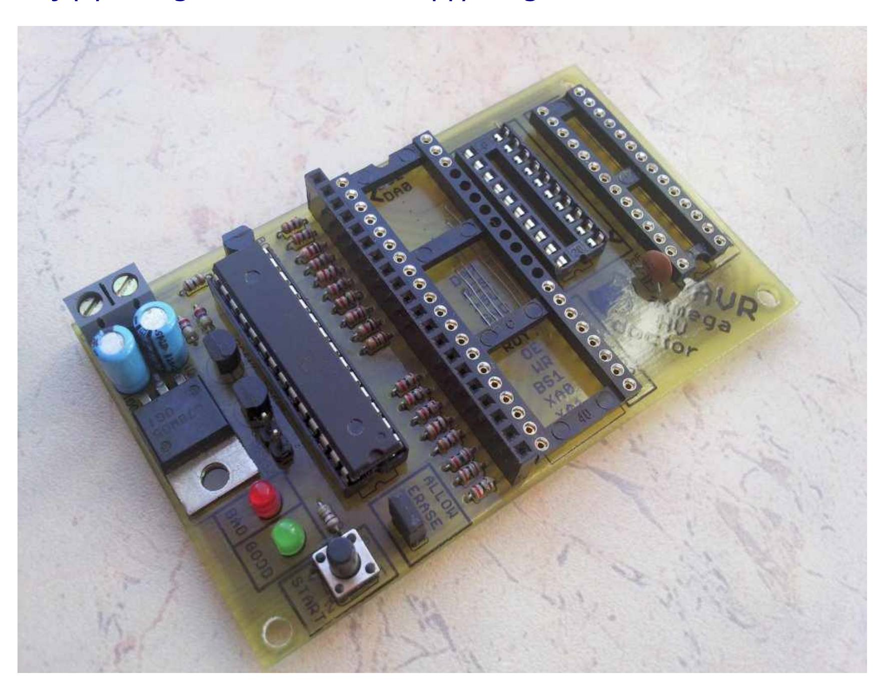
كما ذكرت سابقاً الإعدادات الخاطئة للفيوزات قد تتسبب في عدم إمكانية برمجة المُتحكِم مرة أخرى وبذلك قد تخسر المُتحكِم، ومع ذلك هناك خبر جيد وهو أنه يتوفر نوعين من الحلول لهذا الأمر.

- **الأول:** هو شراء و استخدام الـ High Voltage programmers غالية الثمن مثل AVR Dragon حيث يمتلك القدرة على التنقيح والبرمجة عالية الجهد لتصليح الفيوزات.

- **الثاني:** بناء دائرة AVR Fuse Doctor وهي من الدوائر الرائعة التي تستخدم في معالجة الفيوزات بتكلفة منخفضة جداً والتي جربتها بنفسي وكانت رائعة كما أنه يمكنك صناعتها بنفسك بنحو 5 دولار فحسب.

جميع التصميمات لهذه الدائرة يمكنك أن تجدها على الموقع التالي:


http://mdiy.pl/atmega-fusebit-doctor-hvpp/?lang=en



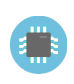
تتوفر تصميمات مشابه أيضاً لنفس الدائرة لكن باستخدام مُتحكِمات أخرى بدل من atmega8 مثل ATTiny Fuse doctor

http://www.instructables.com/id/AVR-Attiny-fusebit-doctor-HVSP

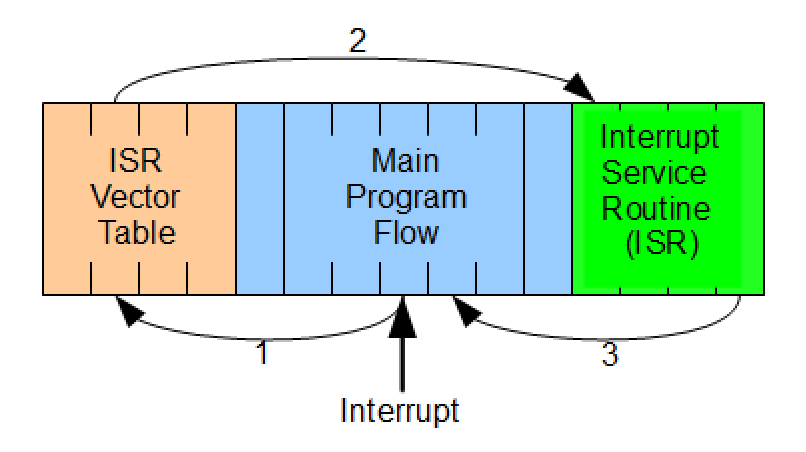
الفصل السابع

" إن أهم يومين يمران على الإنسان هما يوم ولادته
واليوم الذي يدرك فيه لماذا ولد "

مارك توين – كاتب أمريكي

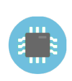


Interrupt المُقاطعة .7



سنتعرف في هذا الفصل على كيفية تشغيل المقاطعات الخارجية External Interrupts وفائدة هذه الخاصية الرائعة التي تتيح صناعة تطبيقات ذات استجابة عالية السرعة للأحداث الخارجية.

- ✔ مقدمة عن مفهوم المقاطعة
- ✔ المثال الأول: تشغيل المقاطعة INT0
- ✔ أنواع الإشارات الرقمية Logic, Falling 7 Raising Edges
- ✔ المثال الثاني: تشغيل INT0 مع INT1



7.1 مقدمة عن المُقاطعة The interrupt

في الكثير من الأنظمة المدمجة نجد بعض الوظائف التي تتطلب استجابة فائقة السرعة لحدث معين. لذا تمت مراعاة هذا الأمر في معظم المعالجات والمُتحكِمات الدقيقة (حتى القديمة منها) حيث تم إضافة تقنية المُقاطعة interrupt وهي عبارة عن طرف دخل input pin أو حدث برمجي يتسبب في جعل المعالج **يتوقف عن ما يفعله الآن ويستجيب للحدث المُسبب للمقاطعة ويعالجه بسرعة ثم يعود مرة أخرى لما كان يفعله.**

مثلاً نجد أن النظام المدمج داخل وحدة تحكم السيارة يعمل على إدارة الوقود وعرض سرعة الحركة ومع ذلك في حالة حدوث اصطدام بجسم ما نجد أن النظام يستجيب بسرعة عالية (بالرغم أنه كان مشغول بمعالجة الوقود والسرعة). تحدث هذه الاستجابة فائقة السرعة بسبب أن الحساسات المسؤولة عن الاصطدام يتم توصيلها على أطراف دخل المُقاطعة وتسمى هذه الأطراف External interrupts.

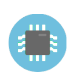
الحقيقة أنه هناك أنواع كثير للمقاطعة (بعضها داخلي وبعضها خارجي) سنتحدث في هذا الفصل عن النوع الخارجي فقط External interrupt وسيتم شرح بعض الأنواع الأخرى على مدار الفصول التالية مثل مقاطعة الـ ADC (في الفصل التالي) ومقاطعة المؤقتات Timer interrupt.

كيف تعمل المُقاطعة الخارجية

في جميع المعالجات والمُتحكِمات الدقيقة يتم تصميم الكود المسؤول عن معالجة المقاطعات بصورة مستقلة تماماً عن البرنامج الرئيسي main program. فنجد دائماً أن برنامج المُقاطعة ويسمى **Interrupt service routing (يُختصر بكلمة ISR)** يكتب في جزء بعيد عن دالة main() فمثلاً يمكنك أن تكتب برنامج main ليقوم بعمل محدد إلى الأبد ثم تكتب برنامج ال ISR ليقوم بوظيفة محددة وسريعة عند تشغيل حساس أو زر معين.

يمتلك المُتحكِم الدقيق من فئة ATmega16/ATmega32 عدد 3 أطراف للمقاطعة الخارجية يمكن توصيلها بأي حساس أو مفتاح رقمي وهذه الأطراف هي.

INT0 (pin 2 on port D)
INT1 (pin 3 on port D)
INT2 (pin 2 on port B)



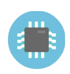
(XCK/T0) PB0	1	40	PA0 (ADC0)
(T1) PB1	2	39	PA1 (ADC1)
(INT2/AIN0) PB2	3	38	PA2 (ADC2)
(OC0/AIN1) PB3	4	37	PA3 (ADC3)
(SS) PB4	5	36	PA4 (ADC4)
(MOSI) PB5	6	35	PA5 (ADC5)
(MISO) PB6	7	34	PA6 (ADC6)
(SCK) PB7	8	33	PA7 (ADC7)
RESET	9	32	AREF
VCC	10	31	GND
GND	11	30	AVCC
XTAL2	12	29	PC7 (TOSC2)
XTAL1	13	28	PC6 (TOSC1)
(RXD) PD0	14	27	PC5 (TDI)
(TXD) PD1	15	26	PC4 (TDO)
(INT0) PD2	16	25	PC3 (TMS)
(INT1) PD3	17	24	PC2 (TCK)
(OC1B) PD4	18	23	PC1 (SDA)
(OC1A) PD5	19	22	PC0 (SCL)
(ICP1) PD6	20	21	PD7 (OC2)

أيضاً يمكن اعتبار الطرف RESETأحد أطراف الـ External interrupt.

ملاحظة: الـ **RESET** في عالم المُتحكِمات الدقيقة يختلف قليلاً عن الحاسب الآلي. فمثلاً نجد في الحاسب الآلي (أو الهاتف الجوال) الزر RESET أو Restart والذي يعني **إيقاف تشغيل الطاقة** عن الحاسب ثم إعادة تشغيله. أما في المُتحكِمات الدقيقة الزر RESET هو مقاطعة خارجية تأمر المعالج **أن يترك ما يفعله الآن وينتقل إلى أول أمر في البرنامج المخزن بداخله** مع تصفير جميع المُسجِلات ووحدات الذاكرة (وهذا يحاكي إعادة تشغيل المُتحكِم الدقيق) ومع ذلك تظل الكهرباء **متصله بالمتحكم ولا يتم فصلها**. والسر وراء تصميم الـ RESET بهذه الطريقة هو أن إعادة فصل وتوصيل الكهرباء بالمتحكم قد يستغرق 60 مللي ثانية وهذا رقم كبير نسبياً في التطبيقات التي تحتاج استجابة سريعة بينما المُقاطعة يتم تشغيلها في أقل من 1 ميكروثانية (يعني أسرع بنحو 60,000 ضعف من فصل الكهرباء وإعادة توصيلها).

سنتعرف بالتفصيل على هذا الأمر في الفصل الخاص بإدارة الطاقة والفيوزات.

عند إدخال إشارة رقمية على هذه الأطراف تحدث المُقاطعة. وعندها يترك المُتحكِم الدقيق البرنامج الرئيسي الذي ينفذه وينتقل إلى برنامج الـ ISR ليقوم بمعالجة المُقاطعة. الكود التالي يمثل التركيب البسيط لـ ISR مع الـ main program.



```
int main()
{
...........
}
ISR(interrupt_type)
{
...........
}
```

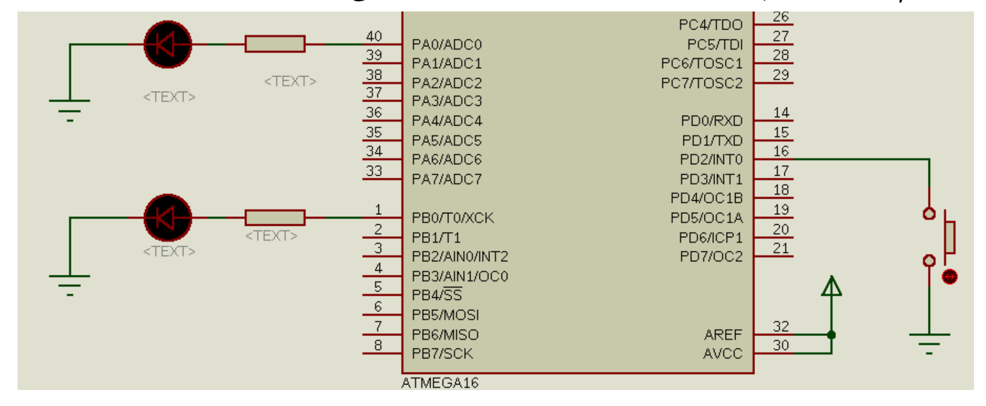
خطوات تفعيل المُقاطعة الخارجية

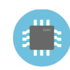
يتم تفعيل المُقاطعة الخارجية بمجموعة من الإعدادات كالتالي:

1. ضبط الأطراف التي ستستخدم للمقاطعة مثل INT0 أو INT1 لتعمل كدخل input
2. ضبط نوع الإشارة الكهربية التي ستسبب المُقاطعة على حسب نوع الحساس أو المفتاح الذي سيولد إشارة المُقاطعة. (انظر للشرح بالأسفل).
3. يتم تفعيل قبول استقبال المُقاطعة على الطرف المطلوب مثل INT0
4. تفعيل قبول استقبال المُقاطعة بشكل عام
5. كتابة البرنامج الخاص بالمُقاطعة ISR

7.2 المثال الأول: تشغيل المُقاطعة INT0

الدائرة التالية عبارة عن 2 دايود ضوئي + مفتاح الدايود المتصل بالطرف PA0 سنقوم بتشغيله بصورة طبيعية ليقوم بعمل Blink كل مئة مللي ثانية. أما الدايود المتصل بالطرف PC0 سيتم تشغيله أو إطفاؤه فقط عند حدوث مقاطعة على الطرف INT0.





الكود

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

int main(void)
{

//ضبط أطراف التوصيل بالدايودات الضوئية
  DDRA |= (1 << PA0);

  DDRB |= (1 << PB0);

// ضبط الطرف الخاص بالمُقاطعة وتشغيل مقاومة الرفع

  DDRD &= ~(1 << PD2);

  PORTD |= (1 << PD2);

// INT0 ضبط نوع إشارة المُقاطعة وتفعيل

  MCUCR |= (1 << ISC01);

  GICR  |= (1 << INT0);

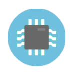
// تفعيل قبول المُقاطعة العامة

  sei();
   while(1)
   {
      PORTA ^= (1 << PA0);

      _delay_ms(100);

   }
   return 0;
}
```

البرنامج الرئيسي



```
ISR(INT0_vect)
{
  PORTB ^= (1 << PB0);
}
```

برنامج المقاطعة

شرح الكود

في بداية الكود قمنا باستيراد المكتبة المسؤولة عن المقاطعات وذلك عن طريق الأمر

```
#include <avr/interrupt.h>
```

هذه المكتبة تحتوي على بعض الأوامر الهامة والتي سنستخدمها في الكود. بعد ذلك بدئنا في الدالة الرئيسية main بضبط الأطراف التي سيتصل بها الدايودات الضوئية وهي الطرفين PA0 و PB0 وذلك عن طريق الأمرين:

```
DDRA |= (1 << PA0);
DDRB |= (1 << PB0);
```

الخطوة التالية كانت ضبط الطرف **PD2** ليعمل كدخل وذلك حتى يتمكن من استقبال إشارة المُقاطعة من المفتاح المتصل به. كما قمنا بتشغيل مقاومة الرفع الداخلية **internal pull up** وذلك حتى نستخدم المفتاح دون الحاجة لتوصيل أي مقاومة إضافية وتم ذلك عن طريق الأمرين:

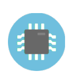
```
DDRD &= ~(1 << PD2);
PORTD |= (1 << PD2);
```

ثم تلى ذلك ضبط نوع المُقاطعة الخارجية ونوع الإشارة الكهربية التي تسبب المُقاطعة. وقبل أن نبدأ في شرح الأوامر الخاصة بهذا الأمر علينا أن نتعرف على بعض الأشياء المتعلقة بالإشارات الكهربية الرقمية.

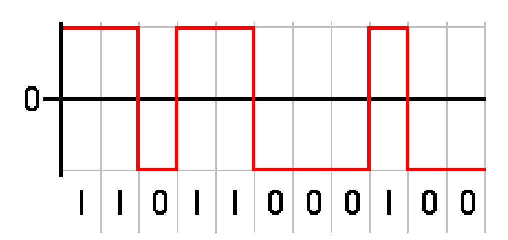
الإشارات الرقمية

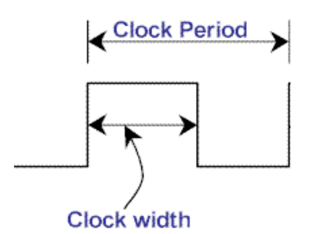
تنقسم الإشارات الكهربية الرقمية إلى نوعين وهما **logic level** و **falling or rising Edge**

النوع الأول وهو المعروف لدى الجميع ويسمى HIGH أو LOW ويعبر عن القيم الرقمية التقليدية 1 & 0. وتكون كل إشارة سواء 0 أو 1 لها زمن محدد يقاس على حسب الـ clock المستخدمة لتشغيل المُتحكِم الدقيق.



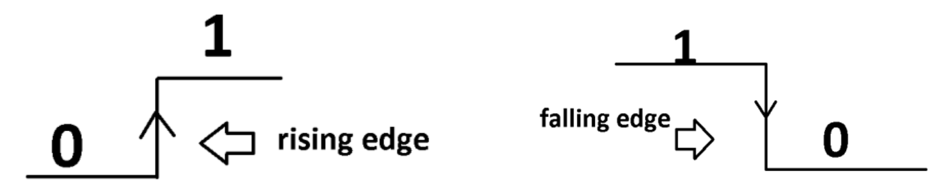
فمثلاً لو كان المُتحكِم يعمل ب clock = 1 Mhz (مليون هرتز) بان زمن النبضة الواحد = 1 ميكروثانية ويكون هذا الزمن هو نفس الزمن المطلوب لعمل اشارة كهربية بقيمة 1 أو صفر. الصور التالية توضح شكل إشارة رقمية مقسمة إلى وحايد وأصفار (حيث يمثل كل مربع رمادي اللون زمن إشارة واحدة).





النوع الثاني من الإشارات الرقمية يسمى "الحواف Edges" والتي تنقسم إلى نوعين وهما الحافة الصاعدة Rising Edge والحافة الهابطة Falling Edge. هذا النوع من الإشارات الكهربية يتميز بأنه فائق السرعة ولا يلتزم بزمن محدد وغالباً ما يحدث في زمن يقاس بالنانو ثانية (جزء من مليار من الثانية).

وتعتبر الحواف الصاعدة هي تحول الإشارة الكهربية من LOW level إلى HIGH level في زمن صغير جداً بينما الحواف الهابطة هي تحول الإشارة الكهربية من HIGH level إلى LOW level.



الصور السابقة تثير تساؤل هام وهو: ألا يعني ذلك أن جميع الإشارات الرقمية تحتوي على falling و rising edges ؟

الإجابة هي **نعم**. والحقيقة أن أي إشارة رقمية تحتوي على حواف عددها يساوي ضعف عدد الأصفار والوحايد والصورة التالية توضح الحافة الهابط والصاعدة مع إشارة LOW ثم HIGH



الأطراف التقليدية التي تعمل كدخل للمتحكم الدقيق لا تستطيع أن تستشعر هذه الحواف سواء كانت الصاعدة أو الهابطة وذلك لأنها تتم في زمن صغير جداً. وهنا تظهر مشكلة خطيرة**.** حيث نجد أن بعض الحساسات يكون الخرج الكهربي الخاص بها سريع جداً لدرجة أن الإشارات الكهربية الناتجة منه تكون في زمن يقاس بالنانو ثانية (مثل الحواف الصاعدة والهابطة) وتمسى هذه الإشارات السريعة بالـ **Electric Impulse** أو **Electric Edges.**

لحل هذه المشكلة قام مصممو المُتحكِمات الدقيقة بصناعة دائرة إلكترونية خاصة تتصل بأطراف المقاطعات وتسمى بالـ **edge detector** (مكتشف الحواف). وتكون هذه الدوائر مسؤولة عن الإحساس بالإشارات الكهربية فائقة السرعة وإبلاغ المُتحكِم بأنه هناك مقاطعة مطلوبة فوراً.

تمتلك مُتحكِمات AVR هذه الدوائر الخاصة على جميع أطراف المقاطعات وبذلك يمكننا أن نجعل المُقاطعة تعمل على جميع المفاتيح أو الحساسات فائقة السرعة. حيث يمكن ضبط المُقاطعة أن تعمل إما بإشارة تقليدية 1 & 0 أو عن طريق إشارة سريعة **Edge.**

يتحكم في هذا الأمر المُسجِل MCUCR والذي يحتوي على مجموعة من البتات مسؤولة عن تحديد نوع إشارة المُقاطعة وتسمى **ISC**xx (يتم استبدال xx برقمي 0 و 1 كما سنرى في الشرح التالي). يمكنك الوصول لشرح هذا المُسجِل في الصفحة رقم 68 من دليل البيانات للمتحكم ATmega16.

MCU Control Register – MCUCR

The MCU Control Register contains control bits for interrupt sense control and general MCU functions.

Bit	7	6	5	4	3	2	1	0	
	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- تتحكم البت **ISC00** و **ISC01** في إعدادات الإشارة الخاصة بطرف المُقاطعة **INT0**
- تتحكم البت **ISC10** و **ISC11** في إعدادات الإشارة الخاصة بطرف المُقاطعة **INT1**

عند تغير قيم هذه البتات يمكننا تحديد نوع إشارة المُقاطعة المطلوبة وذلك تبعاً للجدول التالي (يمكنك أن تجده في الصفحة رقم 68 من دليل بيانات ATmega16)

Table 35. Interrupt 0 Sense Control

ISC01	ISC00	Description
0	0	The low level of INT0 generates an interrupt request.
0	1	Any logical change on INT0 generates an interrupt request.
1	0	The falling edge of INT0 generates an interrupt request.
1	1	The rising edge of INT0 generates an interrupt request.

- **الخيار الأول:** هو ترك قيمة ISC00 و ISC01 بصفر وهذا سيجعل المُقاطعة تعمل عند إدخال إشارة من نوع LOW على الطرف INT0 (معنى ذلك أنه طالما الطرف INT0 يدخل إليه إشارة HIGH فلن تعمل المُقاطعة).

- **الخيار الثاني:** جعل قيمة ISC00 تساوي 1 بينما قيمة ISC01 تساوي صفر وهذا سيجعل المُقاطعة تعمل عند حدوث أي تغيير منطقي logic change على الطرف INT0 ومعنى ذلك أنه إذا أدخلت إشارة HIGH شرط أن تكون الإشارة السابقة LOW أو العكس ستعمل المُقاطعة.

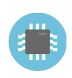
- **الخيار الثالث:** جعل قيمة ISC00 تساوي 0 بينما قيمة ISC01 تساوي 1 وهذا سيجعل دائرة الـ edge detector تعمل على التقاط أي إشارة كهربية هابطة وتشغيل المُقاطعة.

- **الخيار الرابع:** جعل قيمة ISC00 وذلك ISC01 تساوي 1 وهذا سيجعل دائرة الـ edge detector تعمل على التقاط أي إشارة كهربية صاعدة وتشغيل المُقاطعة.

والآن نعود إلى الكود مرة أخرى سنجد نجد الأمر

```
MCUCR |= (1 << ISC01);
```

والذي وضع الرقم 1 داخل البت ISC01 وبما أن البت ISC00 تساوي صفر بصورة افتراضية



فإن هذا الأمر سيجعل المُقاطعة تعمل عند الحافة الهابطة. ولقد اخترت هذا النوع من الإشارات لأن المفتاح المتصل بالطرف INT0 يعمل مع المقاومة الداخلية لهذا الطرف مما يعني أنه عند الضغط على المفتاح سيتحول الطرف INT0 من الحالة HIGH إلى الحالة LOW وهو ما يوازي الحافة الهابطة. أيضاً كان يمكن أن نترك كلا ISC00 و ISC01 بقيمة صفر وهذا سيجعل المُقاطعة تعمل عند الضغط على المفتاح لفترة زمنية قصيرة (1 ميكروثانية على الأقل).

بعد الانتهاء من ضبط نوع إشارة المُقاطعة علينا أن نخبر المُتحكِم الدقيق بأننا نريد تفعيل استقبال إشارات المُقاطعة على الطرف INT0 ويتم ذلك عن طريق التلاعب بالبتات الخاصة بالمسجل GICR (يمكنك الوصول لهذا المُسجِل في الصفحة 69 بدليل البيانات ATmega16).

General Interrupt Control Register – GICR

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	INT2	–	–	–	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

يمتلك هذا المُسجِل 3 بتات هامة جداً وهي INT0 و INT1 و INT2 كل بت من الثلاثة تتحكم في تفعيل استقبال المُقاطعة على أحد الأطراف فمثلاً فعندما نضع القيمة 1 داخل البت INT0 فهذا يعني أن المُتحكِم سيبدأ استقبال إشارات المُقاطعة على الطرف INT0 وهكذا ..

لذا استخدمنا الأمر التالي لأخبار المُتحكِم أن يبدأ تفعيل المُقاطعة INT0

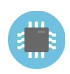
```
GICR  |= (1 << INT0);
```

والآن يأتي الأمر

```
sei();
```

هذا الأمر مسؤول عن تفعيل استقبال طلبات المُقاطعة بشكل عام للمتحكم الدقيق **وبدون كتابته فلن تعمل أي مقاطعة مهما كانت** وذلك حتى وإن كنت قد كتبت جميع أوامر ضبط المُقاطعة. ويعتبر استخدام هذا الأمر مع الأمر **cli** أحد الوسائل في التحكم في سير البرامج الهامة التي لا يمكن مقاطعتها.

وأخيراً نأتي لنهاية الدالة الرئيسية main حيث سنقوم بجعل المُتحكِم يشغل ويطفئ الدايود الضوئي المتصل على الطرف PA0 كل 100 مللي ثانية إلى الأبد.



```
while(1)
{
   PORTA ^= (1 << PA0);
   _delay_ms(100);
}
```

بعد الانتهاء من الدالة الرئيسية يأتي دور **دالة معالجة المُقاطعة ISR** حيث نجد أن هذه الدالة مكتوبة بالأسلوب التالي:

```
ISR(INT0_vect)
{
PORTB ^= (1 << PB0);
}
```

جميع دوال المُقاطعة المختلفة يتم كتابتها عن طريق التعريف **ISR(interrupt_vector)** ويتم استبدال كلمة interrupt_vector بنوع المُقاطعة المطلوبة وبالنسبة للـ External interrupts هناك 3 أنواع :

INT0_vect
INT1_vect
INT2_vect

فإذا أردنا أن نكتب البرنامج الخاص بالمُقاطعة **INT0** فإننا نكتب

```
ISR(INT0_vect)
{
```

وإذا أردنا أن نكتب البرنامج الخاص بالمُقاطعة **INT1** فإننا نكتب

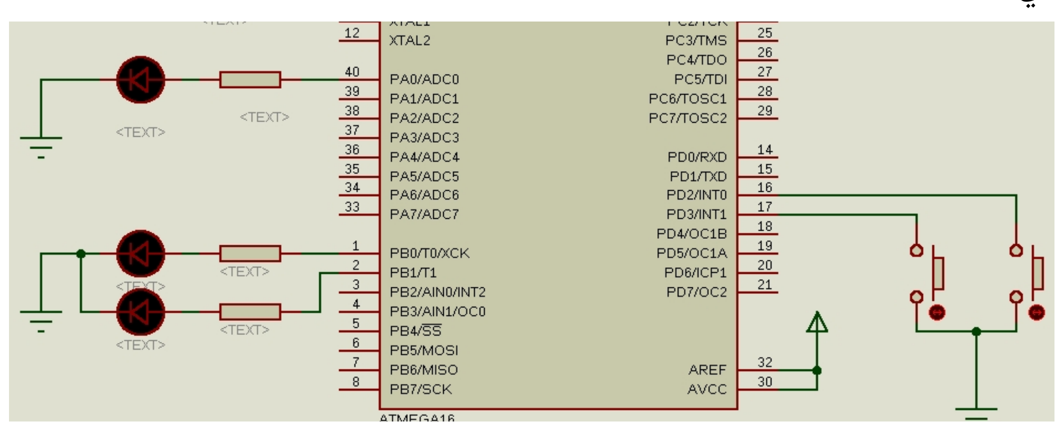
```
ISR(INT1_vect)
{
```

مع وضع الأوامر المطلوبة من المُقاطعة داخل القوسين {}. وفي المثال السابق استخدام أمر يقوم بعكس حالة الطرف PB0 وهذا ما سيجعل الدايود الضوئي يضيء أو ينطفئ عند كل مرة يتم فيها تفعيل المُقاطعة INT0.

```
PORTB ^= (1 << PB0);
```

7.3 المثال الثاني: تشغيل المُقاطعة INT0 مع INT1

في هذا المثال سنقوم بتفعيل كلا المقاطعتين INT0 و INT1 وسنقوم ببناء دائرة مشابهة للمثال السابق باختلاف وجود 2 مفتاح لتفعيل INT0 و INT1 ووجود دايود ضوئي إضافي كما في الصورة التالية:



الكود

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

int main(void)
{
//ضبط أطراف التوصيل بالدايودات الضوئية

  DDRA |= (1 << PA0);

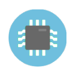
  DDRB |= (1 << PB0) | (1 << PB1);

// INT0 & INT1 ضبط الطرف الخاص بالمُقاطعة وتشغيل مقاومة الرفع

  DDRD &= ~((1 << PD2)|(1 << PD3));

  PORTD |=  (1 << PD2)|(1 << PD3);
```

البرنامج الرئيسي



```
// INT0 & INT1 تفعيل المُقاطعة عند استقبال إشارة من نوع الحافة الهابطة لكل المقاطعتين

   MCUCR |= (1 << ISC01) | (1 << ISC11);
   GICR     |= (1 << INT0)  | (1 << INT1);

   while(1)
   {
      PORTA ^= (1 << PA0);
      _delay_ms(100);
   }
   return 0;
}
// دالة المُقاطعة الأولى //
ISR(INT0_vect)
{
   PORTB ^= (1 << PB0);
}

// دالة المُقاطعة الثانية //
ISR(INT1_vect)
{
   PORTB ^= (1 << PB1);
}
```


شرح الكود

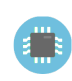
هذا المثال يعتبر مطابق للمثال السابق في نفس الفكرة مع اختلاف تشغيل كلا المقاطعتين INT0 و INT1 بحيث تكون كل مقاطعة مسؤولة عن تشغيل وإطفاء الدايودات الضوئية المتصلة على الأطراف PB0 و PB1.

وكما هو ملاحظ سنجد في نهاية البرنامج دالتين ISR الأولى مسؤولة عن أوامر معالجة INT0 و الثانية مسؤولة عن أوامر معالجة INT1

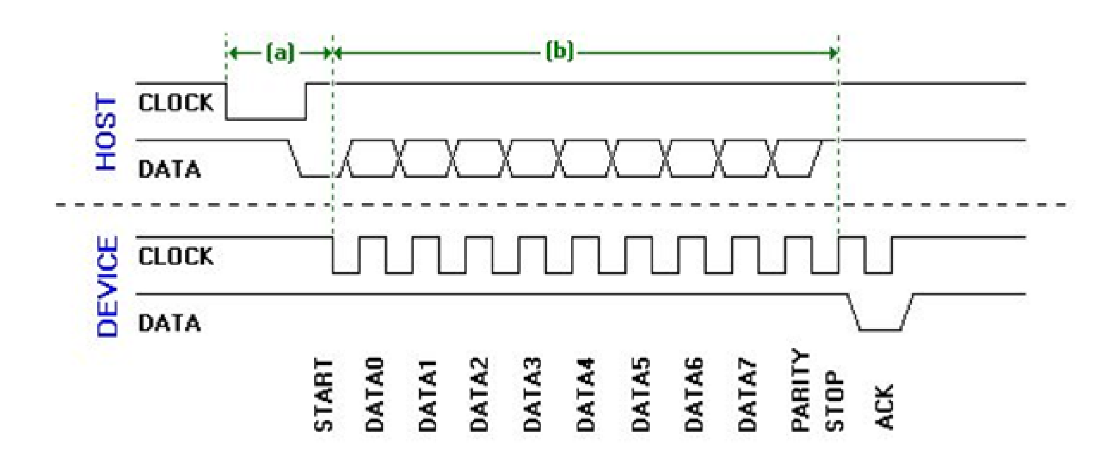
الفصل الثامن

"الرجل الذي يستخدم مهاراته وخياله البناء لمعرفة أقصى ما يمكن أن يقدمه مقابل دولار واحد بدلاً من التفكير في أقل ما يمكن أي يقدمه مقابل نفس الدولار، حتماً سينجح"

هنري فورد - مؤسس شركة فورد للسيارات



8. الاتصال التسلسلي بروتوكول UART



في هذا الفصل سنتعرف على أحد أشهر طرق إرسال البيانات بصورة تسلسلية بين المُتحكِمات الدقيقة والعالم الخارجي وذلك عبر بروتوكول UART والذي يعتبر أشهر بروتوكول معياري لتبادل البيانات.

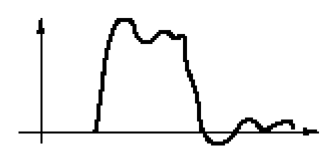
- ✔ مقدمة عن الإتصال التسلسلي
- ✔ الإتصال التسلسلي الغير متزامن
- ✔ تهيئة الـ UART لمتحكمات AVR
- ✔ المثال الأول: تهيئة AVR للعمل كمرسل عبر UART
- ✔ المثال الثاني: تهيئة AVR للعمل كمستقبل عبر UART
- ✔ المثال الثالث: الإرسال و الاستقبال في وقت واحد
- ✔ إرسال السلاسل النصية Strings
- ✔ دوال إضافية

8.1 مقدمة عن الاتصال التسلسلي

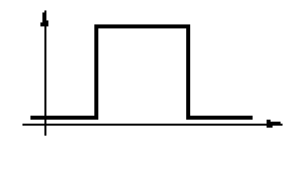
عندما يتواصل المُتحكِم مع العالم الخارجي، فإن إرسال واستقبال البيانات يكون بشكل حزم مكونة من 8 بت(1 بايت). بالنسبة لبعض الأجهزة مثل الطابعات القديمة داخل كابل الـ Parallel port يتم إرسال البيانات من ناقل البيانات 8 بت (bit data bus-8) من الكمبيوتر إلى ناقل البيانات 8 بت في الطابعة.

يعيب هذا الأسلوب في نقل البيانات وجوب أن تكون المسافة بين الجهازين قصيرة. لأن الأسلاك تشوه شكل الإشارات الكهربية مع طول المسافة، كما أن الأسلاك المستخدمة لنقل 8 بت في نفس الوقت يكون سعرها مرتفع.

أيضاً تحدث مجموعة من الظواهر كهربية تسمى "المكثفات الطفيلية Parasitic Capacitance" و "الملفات الطفيلية Parasitic inductance" هذه الظواهر تحدث للوصلات النحاسية المتقاربة من بعضها البعض. وتتسبب في تشويه كبير لشكل الإشارة. الصورة التالية توضح شكل إشارة كهربية على صورة "نبضة pluse" بعد التشويه.



شكل الإشارة بعد التشويه الناتج من المكثفات والملفات الطفيلية



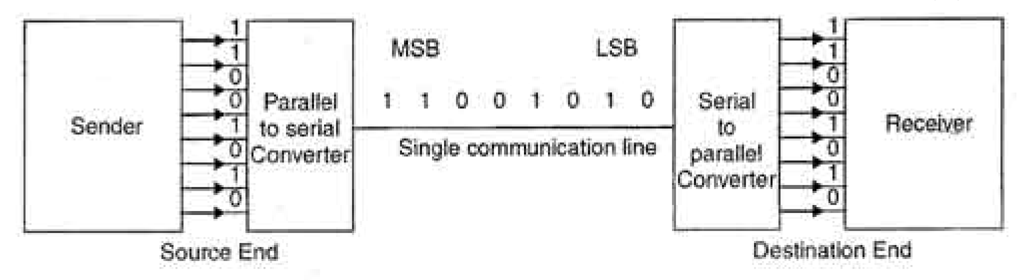
شكل الإشارة الأصلية

لحل هذه المشكلة يتم استخدام الاتصال التسلسلي Serial communication لنقل البيانات بين الأنظمة التي تفصل بينها مسافات كبيرة، وفي وقتنا الحاضر ومع تطور التكنولوجيا أصبح الاتصال التسلسلي أسرع من ذي قبل، فتم تعميمه واستخدامه حاليا في جميع الأجهزة تقريباً بدءاً من الحساسات في الأنظمة المدمجة إلى الحواسيب الشخصية وشبكات الحاسب الآلي.

مبدأ عمل الاتصال التسلسلي UART

تمتلك المُتحكِمات الدقيقة مجموعة من الوسائل التي تمكنك من توصيل البيانات من وإلى المُتحكِم باسلوب تسلسلي ومنها UART – SPI – i2c في هذا الفصل سنتحدث عن الـ UART.

تستخدم تقنية الاتصال التسلسلي طرف (سلك) واحد فقط لنقل البيانات من جهاز لآخر بدلاً من 8 أسلاك كما في حالة الاتصال المتوازي Parallel ولكي يتم إرسال البيانات بشكل تسلسلي يتم أولاً تحويل البيانات من 8 بت Parallel إلى 8 بتات متسلسلة وذلك باستخدام شريحة إلكترونية (متواجدة داخل المُتحكِم الدقيق) تسمى Parallel-in-Serial-out shift register وهو عبارة عن مُسجِل إزاحة يكون دخله 8 بتات parallel وخرجه 8 بتات متسلسلة. وعلى الجانب الآخر، يجب أن يمتلك المُستقبل شريحة أخرى تقوم بعكس هذه العملية وتسمى Serial-in-Parallel-out shift register، لتحويل البيانات مرة أُخرَى إلى 8 بت متوازية.



Serial transmission

> ملاحظة: كلمة بروتوكول Protocol تعني طريقة تنظيم إرسال واستقبال البيانات مثل سرعة البيانات وطريقة ترتيبها وترقيم البيانات المرسلة وكذلك الأطراف المستخدمة لهذا الإرسال والاستقبال

أنواع الإرسال التسلسلي

يمكن نقل البيانات تسلسلياً بپروتوكول UART بطريقتين، لكل منهما مميزات وعيوب وهما:

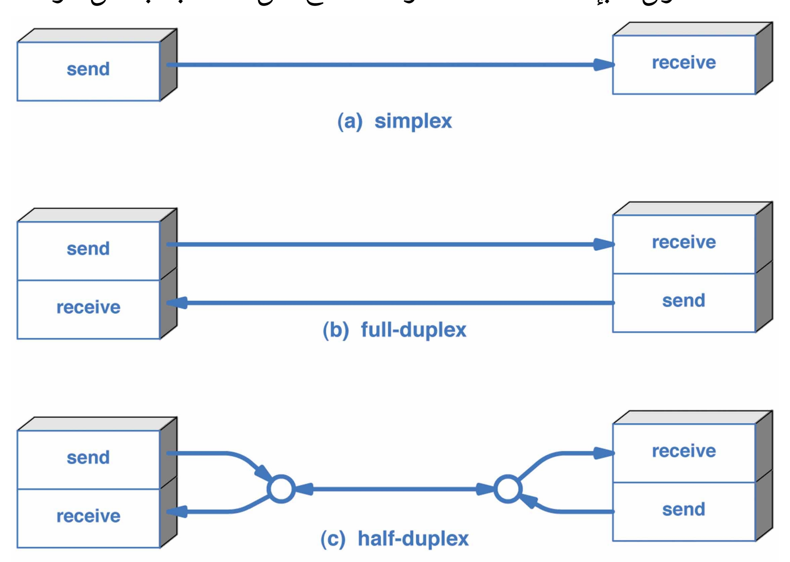
- **الاتصال التسلسلي المتزامن.** **Synchronous**
- **الاتصال التسلسلي الغير متزامن.** **Asynchronous**

يُستخدَم الاتصال المتزامن لنقل كمية من البيانات دفعة واحدة (Block of data)، بينما يُستخدَم الاتصال الغير متزامن لنقل بايت واحد في كل مرة.

ويمكن برمجة المُتحكِم للعمل بإحدى الطريقتين، ولكن البرنامج سيكون طويلا. لذلك تم صناعة دوائر متكاملة يتم دمجها داخل المُتحكِم مخصصة للاتصال التسلسلي، وأصبح يرمز إليها ب UART أي Universal Asynchronous Receiver Transmitter. أو USART أي Universal Synchronous-Asynchronous Receiver Transmitter. وتحتوي مُتحكِمات AVR على USART داخلي.

هناك نوعان للإرسال التسلسلي:

1. **Simplex**: عندما يكون هناك إرسال فقط أو استقبال فقط. مثل: الطابعة، فالكمبيوتر هو الوحيد الذي يرسل البيانات.
2. **Duplex**: عندما يكون هناك قابلية للإرسال والاستقبال، وينقسم إلى نوعين:
3. **Half-duplex**: وذلك عندما تكون هناك القابلية للإرسال والاستقبال ولكن ليس في آن واحد، مثل: جهاز اللاسلكي، عندما تريد التحدث تضغط على الزر وتبدأ في التحدث، والجهاز الآخر يمكنه فقط الاستماع، وعند إزالة يدك من على الزر يمكن للجهاز الآخر إرسال الصوت وأنت يمكنك الاستماع.
4. **Full-duplex**: عندما تكون هناك القابلية للإرسال والاستقبال في آن واحد، مثل: الهاتف المحمول، فبإمكانك التحدث والاستماع لمن تخاطبه بنفس الوقت.

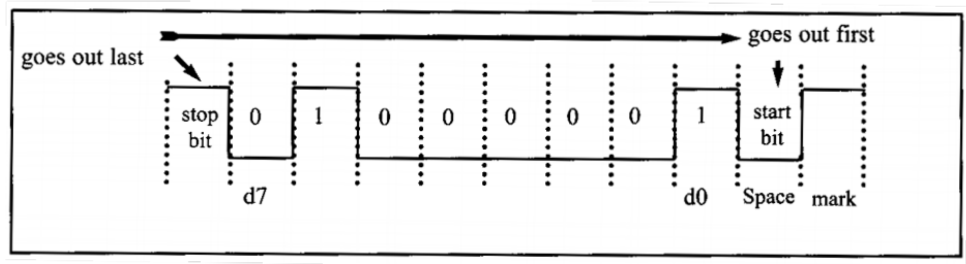


8.2 التسلسلي الغير متزامن Asynchronous

تستقبل البيانات بجهة المستقبل علي هيئة 0 و 1. ولا يمكن معرفة ماهية هذه البيانات إلا عندما يتفق المرسل والمستقبل على مجموعة من القواعد والضوابط " بروتوكول" حول كيفية إرسال البيانات، وكم عدد البتات في كل مرة، ومتى يبدأ الإرسال ومتى ينتهي.

بتات بداية ونهاية الإرسال:

يتم إرسال البايت الواحد بين بت للبداية أُخرَى للنهاية، وهذا يدعى بال " إطار " Frame. بت البداية first bit تكون عبارة عن نبضة واحدة وتكون دائما LOW، بينما بت النهاية يمكن أن تكون نبضة واحدة أو 2 بت وتكون دائما HIGH.
الصورة التالية تمثل مثال على إرسال كود ASCII للحرف A حيث يتم إرسال 10 بتات لكل 1 بايت. 8 بت للبيانات (حرف A نفسه) وبت للبداية أُخرَى للنهاية.



Courtesy of: AVR microcontroller and embedded systems using assembly and C by M.Ali Mazidi

في بعض الأنظمة تضاف بت أُخرَى وتسمى Parity bit، وتستخدم لمعرفة إذا ما كانت البيانات المستلمة صحيحة أم بها خطأ.

معدل إرسال البيانات:Baud rate

يقاس معدل إرسال البيانات في الاتصال التسلسلي ب bps أي bits per second. بت في الثانية. وتعتمد سرعة إرسال البيانات على النظام المستخدم، فقد كانت أجهزة IBM القديمة ترسل البيانات بسرعات تتراوح بين 100 إلى 9600 bps. ومع التطور استطاعت أجهزة المودم إرسال البيانات بسرعة تصل إلى 56kbps .

في الوقت الحالي تدعم معظم المُتحكِمات الدقيقة (بما في ذلك AVR) سرعة أنظمة الإرسال التسلسلية من نوع Asynchronous بحد أقصى 115200 بت في الثانية (نحو 100 كيلوبت في الثانية).

أطراف الإرسال والاستقبال في المُتحكِم ATmega16/32

PDIP

(XCK/T0) PB0	1		40	PA0 (ADC0)
(T1) PB1	2		39	PA1 (ADC1)
(INT2/AIN0) PB2	3		38	PA2 (ADC2)
(OC0/AIN1) PB3	4		37	PA3 (ADC3)
($\overline{SS}$) PB4	5		36	PA4 (ADC4)
(MOSI) PB5	6		35	PA5 (ADC5)
(MISO) PB6	7		34	PA6 (ADC6)
(SCK) PB7	8		33	PA7 (ADC7)
$\overline{RESET}$	9		32	AREF
VCC	10		31	GND
GND	11		30	AVCC
XTAL2	12		29	PC7 (TOSC2)
XTAL1	13		28	PC6 (TOSC1)
(RXD) PD0	14		27	PC5 (TDI)
(TXD) PD1	15		26	PC4 (TDO)
(INT0) PD2	16		25	PC3 (TMS)
(INT1) PD3	17		24	PC2 (TCK)
(OC1B) PD4	18		23	PC1 (SDA)
(OC1A) PD5	19		22	PC0 (SCL)
(ICP1) PD6	20		21	PD7 (OC2)

الطرف التي تحمل الرمز **RXD تستخدم للاستقبال:** ويتم توصيلها بالطرف الخاصة بالإرسال في المُتحكِم الآخر.

الطرف التي تحمل الرمز **TXD تستخدم للإرسال**: ويتم توصيلها بالطرف الخاصة بالاستقبال في المُتحكِم الآخر.

8.3 تهيئة الـ UART الداخلي لمتحكمات AVR

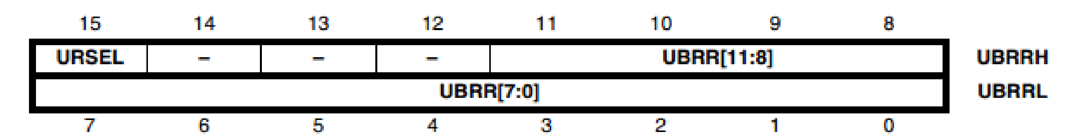
يتم تهيئة ال UART للعمل عن طريق ضبط الإعدادات الخاصة ب: معدل نقل البيانات - عدد بتات الإرسال - عدد بتات النهاية... وغيره من الإعدادات والتي يتم ضبطها عن طريق تغيير قيم المُسجِلات التي تتحكم في ال UART.

يتحكم في ال UART 5 مسجلات وهي:

1- .UBRR [H: L]: USART Baud Rate Register
2- .UCSRA: USART Control and Status Register A
3- .UCSRB: USART Control and Status Register B
4- .UCSRC: USART Control and Status Register C
5- .UDR: USART I/O Data Register

شرح المُسجِلات

UBRR [H: L]



15	14	13	12	11	10	9	8	
URSEL	–	–	–	UBRR[11:8]				UBRRH
UBRR[7:0]								UBRRL
7	6	5	4	3	2	1	0	

وهو عبارة عن مسجلين 8بت UBRRL ويحمل ال 8 بت 1بت القيمة الصغرى من قيمة ال baud rate، والمسجل الآخر هو UBRRH ويحتوى على القيمة العظمى من ال baud rate.

يتم وضع قيمة ال baud rate في البتات من 0 إلى 11.

ملاحظة: بالنسبة لبعض المُسجِلات سيكون الشرح متعلق بالبتات التي سنستخدامها فقط.

UCSRA

7	6	5	4	3	2	1	0	
RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	UCSRA
R	R/W	R	R	R	R	R/W	R/W	

- **البت رقم 7: RXC** هذه البت تصبح 1 عند اكتمال استقبال البايت، وتظل 0 أثناء

الاستقبال.

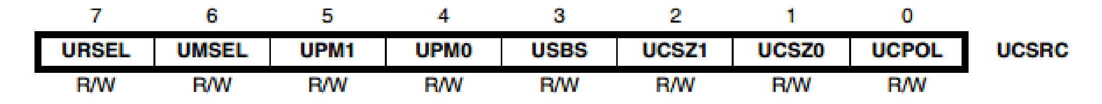
- **البت رقم 6: TXC** هذه البت تصبح 1 عند تمام الإرسال، وتظل 0 أثناء الإرسال.
- **البت رقم 5: UDRE** تكون قيمتها 0 أثناء انشغال المُتحكِم وتصبح 1 عندما يكون جاهزاً لإرسال بيانات أُخرَى.

UCSRB

	7	6	5	4	3	2	1	0
UCSRB	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8
	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W

- **البت رقم 7: RXCIE** عند جعل قيمة هذه البت تساوي 1 ، يتم تفعيل المقاطعة Interrupt الخاصة باستقبال البيانات.
- **البت رقم 6: TXCIE**عند جعل قيمة هذه البت تساوي 1 ، يتم تفعيل المقاطعة Interrupt الخاصة بإرسال البيانات.
- **البت رقم 5: UDRIE**عند جعل قيمة هذه البت تساوي 1 ، يتم تفعيل المقاطعة Interrupt الخاصة بجاهزية المُتحكِم لإرسال أو إستقبال البيانات.
- **البت رقم 4: RXEN**عند جعل قيمة هذه البت تساوي 1 يتم تفعيل إمكانية استقبال البيانات.
- **البت رقم 3: TXEN**عند جعل قيمة هذه البت تساوي 1 يتم تفعيل إمكانية إرسال البيانات.
- **البت رقم 2: UCSZ2** برجاء مراجعة الجدول في الصفحة التالية

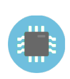
UCSRC



	7	6	5	4	3	2	1	0
UCSRC	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

يحتوي هذا المُسجِل على 2 بت لهما أهمية قصوى وهما البت رقم 2: UCSZ1 وكذلك البت رقم 1: UCSZ0 حيث يستخدمان في تحديد عدد بتات الإرسال في حزمة البيانات الواحدة.

الجدول التالي يوضح كيفية ضبط حجم الحزمة الواحدة من البيانات وذلك بتغير قيم هذه البتات.



UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

سنستخدم في الأمثلة التالية نظام الإرسال بحجم 8 بت (1 بايت). وهذا النظام يعتبر قياسي في معظم المُتحكِمات الدقيقة والأجهزة الإلكترونية المختلفة.

8.4 المثال الأول: تهيئة الـ UART للعمل كمرسل

نسترجع ما شرحناه سابقاً عن بروتوكول الاتصال التسلسلي، وكما أشرنا فإن هناك ما يسمى بمعدل إرسال البيانات والذي يجب أن يتم تحديده للمتحكم، وأيضاً يجب تحديد عدد البتات التي سنرسلها في المرة الواحدة، فالمتحكم قادر على إرسال 5،7،8 أو 9 بتات في المرة الواحدة، هذا بخلاف نبضة البداية ونبضة النهاية. ولكن اتفقنا على اتباع الأنظمة القياسية في تحديد عدد **8 بتات للإرسال**، وهذا ما يجب تحديده للمتحكم.

نبدأ أولا بتحديد معدل نقل البيانات baud rate ويتم تخزين القيمة في المسجلين UBRRH و UBRRL. لنأخذ على سبيل المثال معدل إرسال بيانات يساوي 9600 bps.بمراجعة دليل البيانات للمتحكم يتضح أن تحديد القيمة التي يجب تخزينها بالمسجلين UBRR[H:L] يتم عن طريق العلاقة:

$$UBRR = \frac{f_{OSC}}{16BAUD} - 1$$

- **Fosc** هو تردد المذبذب الدخلي أو الـ " **Crystal** " وسنفترض أننا نشغل المُتحكِم بتردد 16 **ميجا هرتز.**
- **BAUD** هي قيمة معدل إرسال البيانات والتي حددناها سلفاً ب 9600 bps.

> ملاحظة: في هذا المثال سيتم تشغيل المُتحكِم بسرعة 16 ميجا وذلك عبر توصيل كريستالة خارجية 16 ميجا + مكثفات 22 بيكوفاراد، يمكنك مراجعة فصل الطاقة وسرعة التشغيل لتتعرف أكثر على خواص هذا النوع من المذبذبات وطريقة عمله. ولا تنسى أن تضبط برنامج بروتس على محاكاة المُتحكِم بسرعة تشغيل 16 ميجا وذلك عبر الضغط على رمز شريحة ATmega16 مرتين ثم تغير الكريستالة وقيمتها

BOOTRST (Select Reset Vector)	(1) Unprogrammed	Hide All	Cancel
CKSEL Fuses:	(1111) Ext.Crystal High Freq.	Hide All	
Boot Loader Size:	(00) 1024 words. Starts at 0x1C(	Hide All	
SUT Fuses:	(00)	Hide All	
Advanced Properties:			
Clock Frequency	16M	Hide All	

بالتعويض عن هذه القيم في المعادلة السابقة تكون قيمة UBRR تساوي 103.16667 أي بالتقريب تساوي 103. ويتم وضع هذه القيمة كما هو موضح في الكود التالي (هذا الكود يجعل المُتحكِم يرسل قيمة الحرف A بصيغة ASCII كل ثانية).

```
#define F_CPU 16000000
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
  uint16_t UBRR_Value = 103;
  UBRRL = (uint8_t) UBRR_Value;
  UBRRH = (uint8_t) (UBRR_Value >> 8);
  UCSRB = (1<<RXEN) | (1<<TXEN);
  UCSRC |= (3<<UCSZ0);

   while(1)
   {
```

```
    while( ! (UCSRA & (1<<UDRE) ) );
      UDR = 'A';
    _delay_ms(1000);
  }

  return 0;
}
```

شرح الكود

بدايةً قمنا بتعريف متغير 16 بت اسمه UBRR_Value لتخزين القيمة المطلوب كتابتها في المسجلين UBRR[H:L] . ثم أمرنا المُتحكِم بتخزين هذه القيمة في المُسجِل UBRRL ولكن هذا المُسجِل 8 بت فقط. حيث سيتم تخزين أول 8 بت فقط من القيمة. ثم قمنا بتخزين باقي البتات في المُسجِل UBRRH عن طريق الأمر

```
UBRRH = (unsigned char) (UBRR_Value >> 8);
```

وهذا الأمر يقوم بعمل إزاحة لليمين بمقدار 8 بت. ويخزن باقي البتات في هذا المُسجِل.

محتوى المتغير UBRR_Value:

0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	1

ما تم تخزينه بالمسجل UBRRL:

0	1	1	0	0	1	1	1

ما تم تخزينه بالمسجل UBRRH بعد عمل إزاحة لليمين بمقدار 8 بت:

0	0	0	0	0	0	0	0

إلى هنا انتهينا من تحديد قيمة ال baud rate. نأتي الآن لتفعيل إمكانية الإرسال والاستقبال عن طريق الأمر التالي:

```
UCSRB = (1 << RXEN) | (1 << TXEN);
```

بعد هذا الأمر يتبقى شيئ واحد وهو تحديد عدد البتات المرسلة في المرة الواحدة.

```
UCSRC |= (3 << UCSZ0) ;
```

وهذا الأمر يقوم بتعيين عددهم الى 8 بتات.وهو مساوي للأمر

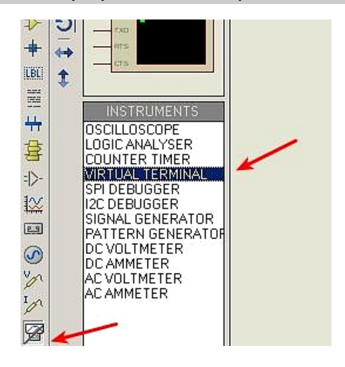
```
UCSRC |= ( (1 << UCSZ2) | (1 << UCSZ0) );
```

والذي يقوم بوضع القيمة 1 في كلا من UCSZ0 & UCSZ2. ولكن للتسهيل استخدمت الأمر بصورته الأولى.

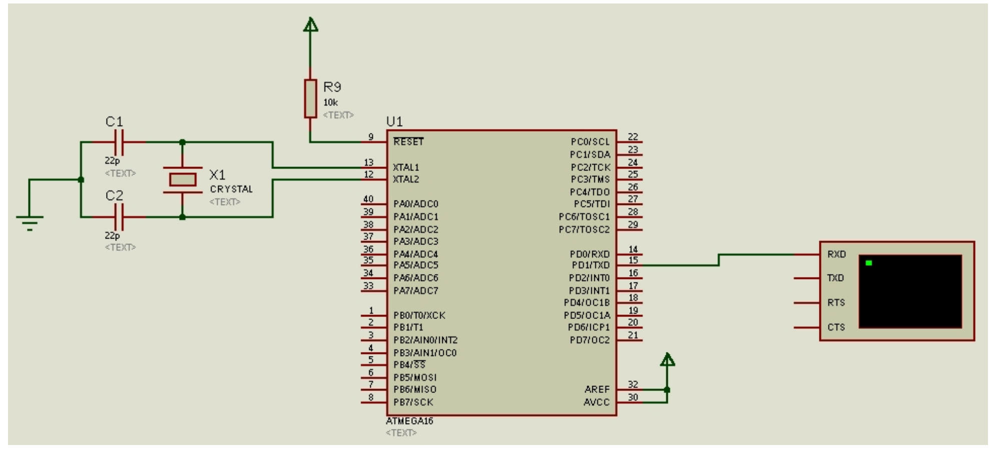
بذلك نكون قد انتهينا من تهيئة ال UART ونستطيع أن نرسل البيانات.ولكن لكي نبدأ الإرسال يجب أن نضع هذه البيانات في المُسجِل UDR وكما ذكرنا سابقاً، يجب أن ننتظر حتى يصبح المُتحكِم جاهزاً لإرسال البيانات لذلك استعنا بالأمر التالي:

```
While (! (UCSRA & (1 << UDRE)));
```

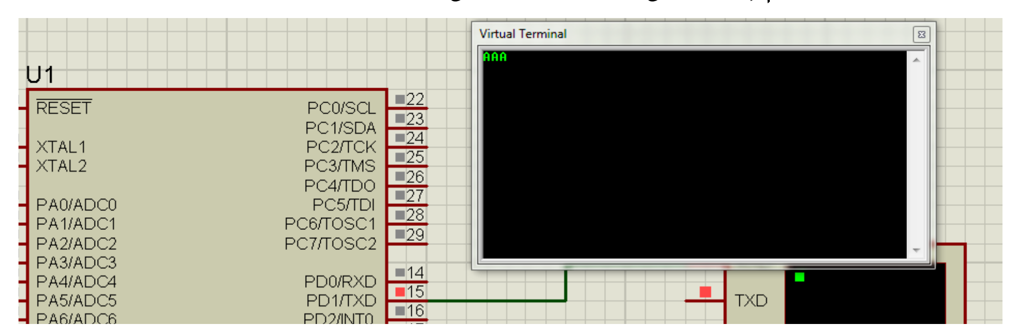
ومعناه أن ينتظر المُتحكِم دون فعل أي شيء طالما البت رقم 5 في المُسجِل UCSRA لا تساوي 1. كما ذكرنا سابقاً عند شرح المُسجِلات أن وجود 1 في هذه البت يدل على أن المُتحكِم جاهز لإرسال البيانات.



الصورة التالية توضح دائرة محاكاة الكود على برنامج بروتس. مع العلم أنه يمكن محاكاة الـ serial port وذلك باستخدام الأداة virtual terminal

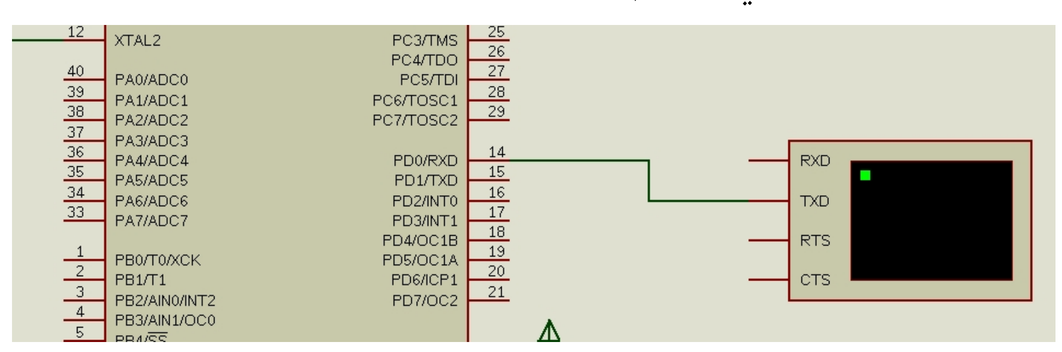


وها هو ذا الحرف A يتم إرساله كل ثانية عند تشغيل المحاكاة:



8.5 المثال الثاني: تهيئة ال UART للعمل كمستقبل

استقبال البيانات عن طريق ال UART يتم بنفس الكود مع عمل تغييرات بسيطة في الدائرة واضافة سطر جديد.(لاحظ أنه في الدائرة الجديدة يتم توصيل الطرف TXDفي ال virtual terminal بالطرف RXD في المُتحكِم الدقيق).




```
#define F_CPU 16000000
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
  uint16_t UBRR_Value = 103;
  UBRRL = (uint8_t) UBRR_Value;
  UBRRH = (uint8_t) (UBRR_Value >> 8);
  UCSRB = (1<<RXEN) | (1<<TXEN);
  UCSRC |= (3<<UCSZ0);

   while(1)

   {

      while (! (UCSRA & (1 << RXC)));

         PORTC = UDR;

   }


   return 0;

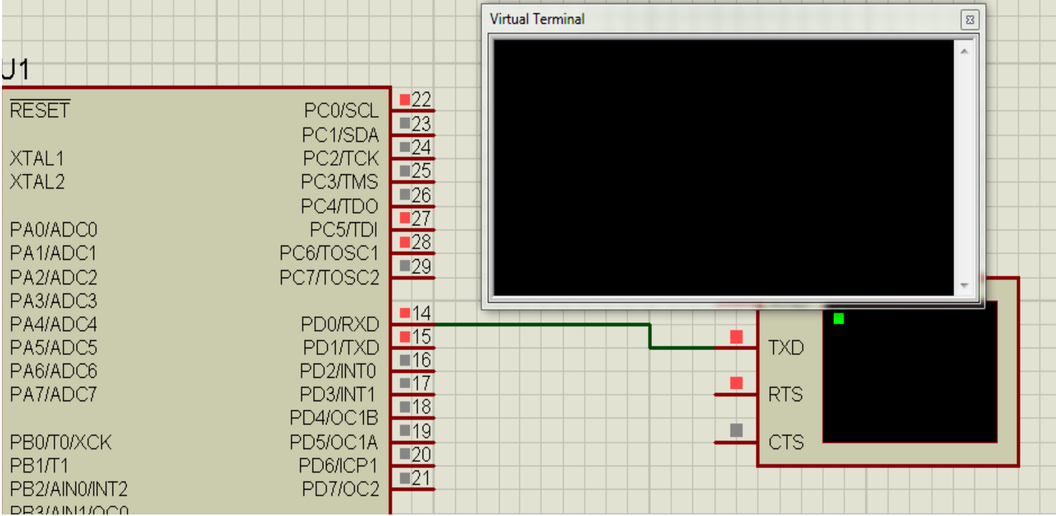
}
```

الاختلاف الوحيد في الكود نجده في الأمر التالي:

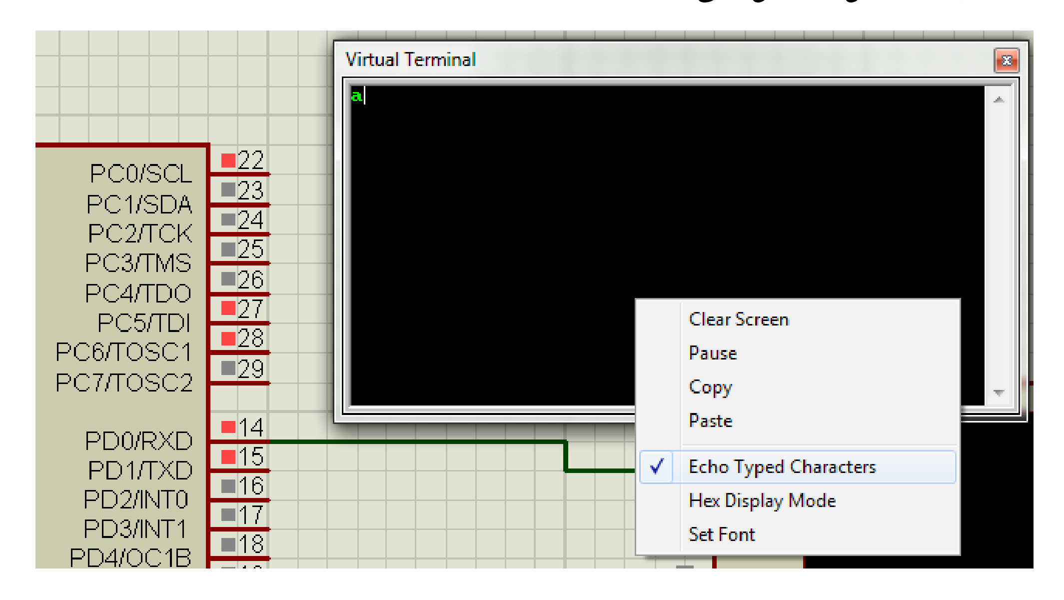
```
While (! (UCSRA & (1 << RXC))); // Waiting for Receiving buffer to be empty.
```

وهذا معناه الانتظار حتى يصبح المُتحكِم جاهزاً للاستقبال. والأمر الذي يليه يقوم بعرض قيمة ما تمت طباعته في نافذة Virtual terminal على PORTC.

شكل التجربة أثناء استقبال الحرف a وكذلك إخراج قيمته (0b01100001) على PORTC.

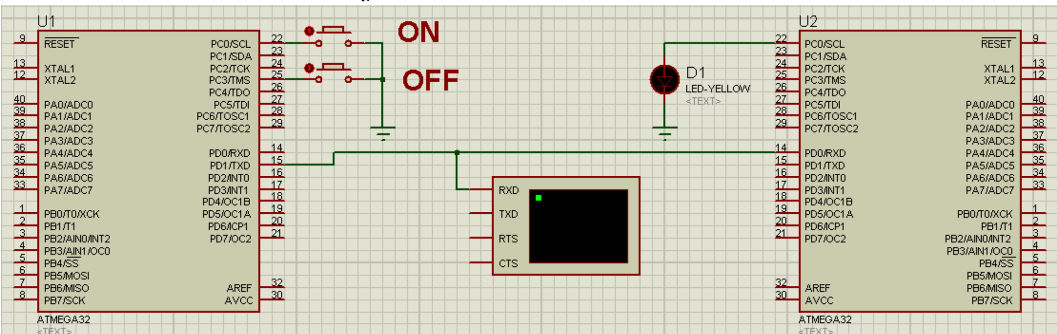


ملاحظة: أثناء كتابة أي حرف على نافذة Virtual terminal لا يتم طباعته على الشاشة ولكن يتم إرساله، وإذا أردت أن يتم طباعته للتحقق مما تضغط عليه، فكل ما عليكم فعله هو الضغط بالزر الأيمن للماوس على النافذة واختيار **Echo Typed Characters.**



8.6 المثال الثالث: الإرسال والاستقبال في وقت واحد

والآن، ما رأيك أن ندمج المثالين السابقين في برنامج واحد. نرسل حرف من مُتحكِم لآخر، وعند استقبال هذا الحرف يضيء المُتحكِم الآخر الدايود الضوئي المتصل به.



لاحظ توصيل الطرف TXD في المُتحكِم بالجانب الأيسر إلى الطرف RXD في المُتحكِم الموجود بالجانب الأيمن.

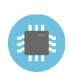
أيضاً سنستخدم زرين، الأول ومكتوب بجانبه ON سيقوم بإرسال الحرف " N "، وعند استقبال هذا الحرف من قبل المُتحكِم الثاني سيقوم بإضاءة الدايود الضوئي. أما الثاني ومكتوب بجانبه OFF فسيقوم بإرسال الحرف " F "، وعند استقباله من قبل المُتحكِم الثاني سيقوم بإطفاء الدايود الضوئي.

هذا هو الكود الخاص بالمتحكم الأول والذي يتولى مهمة إرسال الأحرف عند الضغط على أي من الزرين.

```
#define F_CPU 16000000
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
  DDRC &= ~((1<<PC0) | (1<<PC3));       // ضبط الأطراف لتعمل كدخل
  PORTC |= (1<<PC0) | (1<<PC3);          //تفعيل مقاومة الرفع

  uint16_t UBRR_Value = 103;
```



```
UBRRL = (uint8_t) UBRR_Value;
UBRRH = (uint8_t) (UBRR_Value >> 8);

UCSRB = (1<<RXEN) | (1<<TXEN);

UCSRC |= (3<<UCSZ0);

 while(1)
{
    if(bit_is_clear(PINC,0))
    {
       while(!(UCSRA & (1<<UDRE)));
          UDR = 'N';
       _delay_ms(300);

    }

    if(bit_is_clear(PINC,3))
    {
       while(!(UCSRA & (1<<UDRE)));
          UDR = 'F';
       _delay_ms(300);
    }
}

 return 0;
}
```

السطر الأول يقوم بتحديد الأطراف PC0 و PC3 كمداخل رقمية. من خلال إدخال القيمة 0

في البتات المناظرة لهما في المُسجِل DDRC. والسطر الذي يليه يقوم بتفعيل مقاومة الرفع الداخلية لكل منهما.

ثم نأتي للجملة الشرطية if(bit_is_clear(PINC،0))

هذا السطر يقوم باختبار ما إذا تم الضغط على الزر المتصل ب PC0 أم لا، فإذا تم الضغط على الزر سيقوم المُتحكِم بإرسال الحرف " N ".وكذلك الأمر بالنسبة للزر المتصل ب PC3 ولكن مع فارق أنه يقوم بإرسال الحرف " F ".

أما الكود الخاص بالمتحكم المسئول عن استقبال الأحرف وإضاءة أو إطفاء الدايود الضوئي فهو كالتالي:

```
#define F_CPU 16000000
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
 DDRC |= (1<<PC0);

 uint16_t UBRR_Value = 103;

 char Received;


  UBRRL = (uint8_t) UBRR_Value;

  UBRRH = (uint8_t) (UBRR_Value >> 8);

  UCSRB = (1<<RXEN) | (1<<TXEN);

  UCSRC |= (3<<UCSZ0);


    while(1)

    {

        while (! (UCSRA & (1 << RXC)));

        Received = UDR;


      if(Received == 'N')

        PORTC |= (1<<PC0);
```

```
    if(Received == 'F')
      PORTC &= ~(1<<PC0);
  }

  return 0;
}
```

شرح الكود

في هذا البرنامج قمنا بإنشاء متغير من نوع Character ويدعى Received وقمنا بتخزين ما يتم استقباله في هذا المتغير، ثم يقوم المُتحكِم باختبار محتوى هذا المتغير بجملتين شرطيتين، فإذا كان محتواه مساوياً للحرف " N " قام بإضاءة الدايود الضوئي المتصل ب PC0، وإذا كان محتواه مساوياً للحرف " F " قام بإطفاء الدايود الضوئي.

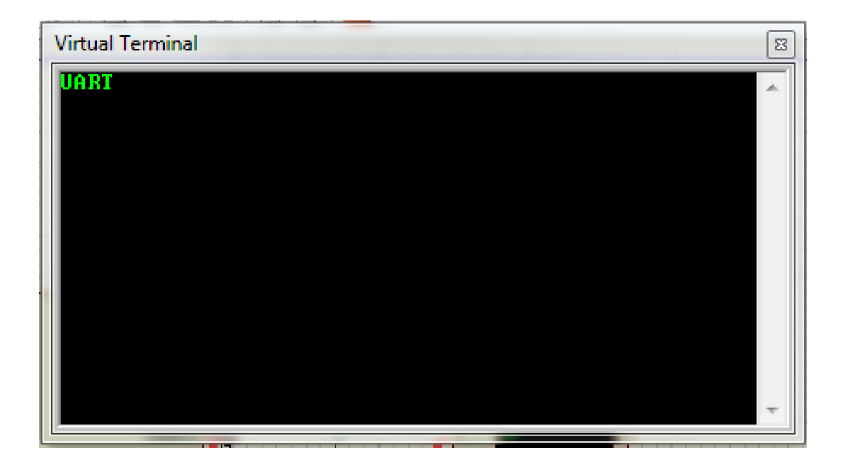
8.7 إرسال مجموعة بيانات مثل السلاسل النصية

قد يتبادر إلى ذهنك، ماذا أفعل إذا أردت إرسال كلمة أو جملة؟ ماذا أفعل إذا أردت إرسال قيمة متغير؟ وماذا أفعل لكي استطيع استقبال كلمة أو جملة؟ لنفترض أننا نحاول إرسال كلمة " UART " سنجد أنه هناك طريقتين لذلك.

الطريقة الأولى: إرسال حروف متتالية

```
while(!(UCSRA & (1<<UDRE)));
  UDR = 'U';                          // إرسال حرف U
while(!(UCSRA & (1<<UDRE)));
  UDR = 'A';                          // إرسال حرف A
while(!(UCSRA & (1<<UDRE)));
  UDR = 'R';                          // إرسال حرف R
while(!(UCSRA & (1<<UDRE)));
  UDR = 'T';                          // إرسال حرف T
```

وسيتم إرسالها.



بالرغم أن هذه الطريقة تصلح بالتأكيد لكنها ليست فعالة وتتطلب كتابة الكثير من الأكواد البرمجية (مما يعني استهلاك المزيد من ذاكرة الـ ROM للمتحكم الدقيق) كما أنه في حالة العبارات الطويلة سيؤدي ذلك لاستهلاك ضخم للذاكرة.

الطريقة الثانية: استخدام المؤشرات

ولكن هناك طريقة أُخرَى. وهو استخدام ال Pointer فالكلمة مكونة من عدد من الأحرف بجانب بعضها. لذا يمكن استبدال الكود السابق بهذا الكود.

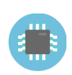
```
char *word = "UART";
while(*word > 0)
{
        while(!(UCSRA & (1<<UDRE)));
        UDR = *word++;

}
```

شرح الكود

في هذا الكود استخدمنا مؤشر يشير إلى بداية الكلمة. ومن أساسيات علم الكمبيوتر أن أي String يحتوي آخره على الرقم 0 ويدعى " null character " لذلك استخدمنا الحلقة التكرارية while(*word > 0) أي طالما أنه يشير إلى شيء أكبر من ال 0 ستستمر الحلقة بالتكرار.

الأمر UDR = *word++ يقوم بإرسال الحرف الذي يشير إليه المؤشر حالياً، ثم يقوم بزيادة



المؤشر ليشير الحرف التالي.حتى يصل إلى نهاية الكلمة فيشير إلى الرقم 0 الذي يتواجد بنهاية أي String، فلا يتحقق شرط الحلقة التكرارية وينتهي تنفيذها.

حسناً، إلى هنا كلما اردنا استخدام ال UART أصبح لزاماً علينا كتابة الأسطر الخاصة بتهيئة الـ UART وأيضاً عند إرسال حرف أو كلمة، يجب كتابة الأسطر الخاصة بذلك. ولكن ما رأيك بجعل الكود أكثر قابلية للاستخدام المتكرر.

لفعل ذلك يجب علينا أن نستخدم الدوال، لأنها تُسهل الأمر كثيراً. ونضع بداخل كل دالة مجموعة الأوامر التي ستنفذها هذه الدالة.مثال على ذلك. الدالة الخاصة بتهيئة ال UART.

```
void UART_init()
{
  uint16_t UBRR_Value = 103;
  UBRRL = (uint8_t) UBRR_Value;
  UBRRH = (uint8_t) (UBRR_Value >> 8);
  UCSRB = (1<<RXEN) | (1<<TXEN);
  UCSRC |= (3<<UCSZ0);
}
```

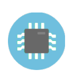
نكتب الدالة السابقة قبل دالة main. وبداخل دالة ال main نقوم باستدعائها لتنفيذ الأوامر التي بداخلها عن طريق الأمر التالي:

```
UART_init();
```

كما هو موضح في الكود التالي (إعادة كتابة المثال الأول ولكن بصورة أفضل):

```
#define F_CPU 16000000
#include <avr/io.h>
#include <util/delay.h>

void UART_init()
{
  uint16_t UBRR_Value = 103;
  UBRRL = (uint8_t) UBRR_Value;
  UBRRH = (uint8_t) (UBRR_Value >> 8);
  UCSRB = (1<<RXEN) | (1<<TXEN);
  UCSRC |= (3<<UCSZ0);
```



```
}
int main(void)
{
 UART_init() ;

  while(1)
  {
    while( ! (UCSRA & (1<<UDRE) ) );
      UDR = 'A';
    _delay_ms(1000);
  }

  return 0;
}
```

والآن استرجع ما ذكرناه في شرح بروتوكول ال UART عن معدل نقل البيانات، وأن هذه السرعة يمكن أن تتغير، أليس من الأفضل جعل الدالة UART_init بإمكانها أن تحدد سرعة نقل البيانات عن طريق أن نمرر هذه السرعة إلى الدالة عند استدعائها؟

تذكر أنه يمكن تحديد هذه السرعة من العلاقة السابق ذكرها،

$$UBRR = \frac{f_{OSC}}{16BAUD} - 1$$

إذاً يمكننا كتابة المعادلة التالية لإيجاد القيمة المراد تخزينها داخل المُسجِل UBRR:

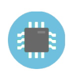
```
uint16_t UBRR_Value = lrint ( (F_CPU / (16L * baud_rate) ) -1);
```

حيث:

F_CPU: تردد المذبذب الذي يعمل عليه المُتحكِم.

Baud_rate: متغير ندخل به قيمة سرعة إرسال البيانات مثل: 9600.

Lrint: هي دالة تقوم بتقريب الناتج إلى أقرب رقم صحيح، ولاستدعاؤها لابد من تضمنيها باستخدام **include** ثم اسم الملف **math.h** في بداية البرنامج.



فيصبح شكل الدالة كالتالي:

```
#define F_CPU 16000000
#include <avr/io.h>
#include <util/delay.h>
#include <math.h>                    //استدعاء مكتبة الحساب

void UART_init()
{
  uint16_t UBRR_Value = lrint ( (F_CPU / (16L * baud_rate) ) -1);
  UBRRL = (uint8_t) UBRR_Value;
  UBRRH = (uint8_t) (UBRR_Value >> 8);
  UCSRB = (1<<RXEN) | (1<<TXEN);
  UCSRC |= (3<<UCSZ0);
}
```

ويتم استدعائها بداخل ال main بالشكل التالي:

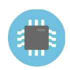
```
int main(void)
{
 UART_init(9600) ;
.........
.........
```

8.8 دوال إضافية

أليس هذا أسهل من ذي قبل؟ إذاً هيا لنفعل المثل ونكتب دالة خاصة بإرسال حرف، وأُخرَى خاصة بإرسال كلمة أو جملة وأُخرَى خاصة باستقبال حرف وواحدة خاصة باستقبال كلمة أو جملة.

- **الدالة الخاصة بإرسال حرف**

```
void UART_send_char(char data)
{
   while( ! (UCSRA & (1<<UDRE) ) );
     UDR = data;
}
```



- **الدالة الخاصة باستقبال حرف**

```
char UART_receive_char()
{
   while (! (UCSRA & (1 << RXC) ) );
          return UDR;
}
```

- **الدالة الخاصة بإرسال String سلسلة حروف**

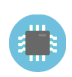
(لاحظ أن هذه الدالة تعتمد على دالة إرسال حرف واحد).

```
void UART_send_string(char *data)
{
    while(*data > 0)
          UART_send_char(*data++);
     UART_send_char('\0');
}
```

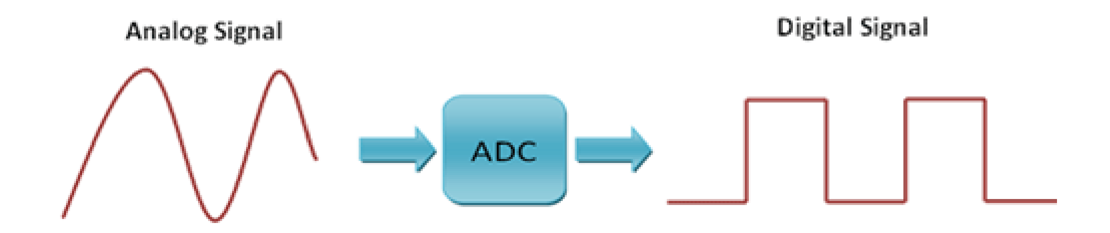
الفصل التاسع

" أن تتعثر فهذا يعني أنك تسير في الطريق، فلم أسمع بأحد يتعثر وهو لا يتحرك "

تشارلز كيترينج - مهندس ومخترع أمريكي



9. المُحَول التناظري-الرقمي ADC



في هذا الفصل سنتعرف على كيفية قراءة الجهود الكهربية المتغيرة Analog وتحويلها إلى قيم رقمية وذلك باستخدام المحول التناظري-الرقمي المدمج داخل مُتحكِمات AVR.حيث يمكن استغلال هذا المحول في قراءة الحساسات التناظرية أو أي عنصر إلكتروني له خرج كهربي متغير.

- ✔ مقدمة عن المحول التناظري – الرقمي ADC
- ✔ طريقة عمل الـ ADC
- ✔ تركيب الـ ADC داخل المُتحكِم ATmega16
- ✔ مثال: قراءة جهد متغير باستخدام مقاومة متغيرة
- ✔ الحسابات الرياضية الخاصة بالـ ADC

9.1 مقدمة عن المحول التناظري-الرقمي ADC

الإشارات الكهربية في العالم الخارجي ليست مقتصرة فقط على الإشارات الرقمية Digital بل على العكس العديد من الأجهزة و الحساسات الإلكترونية تصدر اشارت تماثلية Analog فمثلاً نجد أن معظم الحساسات المتوفرة في الأسواق يمكنها تحويل كمية فيزيائة معينة مثل (درجة الحرارة - الرطوبة - ضغط - تركيز مادة كيميائية ..الخ) إلى فرق جهد كهربي يتغير بتغير هذه القيم الفيزيائية.

بصورة طبيعية لا تستطيع الحواسيب أو المُتحكِمات الدقيقة أن تفهم الاشارات التماثلية فكل ما تفهمه هذه الحواسيب هي الاشارات الرقمية مثل 1 و 0 أو HIGH و LOW. لذا قام مصممو الإلكترونيات بصناعة شريحة خاصة تسمى بالمحول التناظري الرقمي Analog to Digital Converter وتختصر بكلمة ADC هذه الشريحة يمكن شرائها بصورة مستقلة وتوصيلها مع المُتحكِم الدقيق.

لحسن حظنا نجد أن مصممي مُتحكِمات AVR قامو بدمج هذه الشريحة بصورة جاهزة للعمل على معظم مُتحكِمات عائلة AVR mega وبعض مُتحكِمات عائلة ATTiny. وسيناقش هذا الفصل طريقة تشغيل الـ ADC المدمج بعائلات AVR.

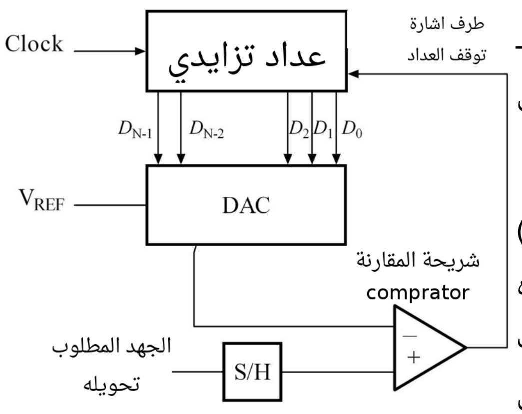
كيف يعمل المُحول التناظري الرقمي



تحتوي معظم المُتحكِمات الدقيقة (من مختلف الشركات) ومنها AVR على ADC من نوع **successive approximation adc** هذا المحول يعمل بطريقة مشابهه **"للميزان القديم"**. تخيل معي أن لديك أحد هذه الموازين القديمة ذات الكفتين. ولنفترض أنك أردت أن تعرف وزن ثمرة بطيخ. فما الذي ستفعله؟

في البداية ستضع ثمرة البطيخ في أحد كفوف الميزان، ثم تدريجياً تبدأ باضافة وزن معروف مسبقاً في الكفة الأخرى فمثلاً قد تضع رُبع كيلو جرام (250 جرام) ثم تنظر إلى الميزان **وتقارن ارتفاع الكفتين**، إذا لم تجد أن الكفتين قد تساوياً ستضيف رُبع كيلو جرام أخرى وهكذا .. **يتم التوقف عن وضع المزيد من الوزن عندما تقترب كفتي الميزان من بعضها.**

بعد أن تتساوى الكفتان الخطوة التالية هي معرفة **عدد الأوزان** التي تم وضعها في كفة القياس (مثلاً قد نجد أنه هناك 3 قطع وزن من نوع رُبع كيلو جرام) ومنها يمكننا أن نحسب وزن الثمرة والذي يساوي 3 × رُبع كيلوجرام = 750 جرام.



الخطوات السابقة تمثل نفس طريقة عمل الـ ADC . حيث نجد أن الـ ADC يحتوي على مجموعة مكونات تعمل مثل الميزان.

المُكَوِّن الأول يسمى دائرة التقطيع (أخذ العينة) وتسمى sample & hold (S/H) circuit. هذه الدائرة تأخذ عينة من فرق الجهد المطلوب تحويله للصورة الرقمية وتدخلها للجزء التالي من ال ADC.

المُكَوِّن الثاني يسمى بـ analog comparator وهو شريحة إلكترونية **تماثل الميزان بالضبط** وتمتلك طرفان للمقارنة. حيث تقوم بالمقارنة بين فرق جهد مطبق على طرفها الأول (والذي يتم الحصول عليه من دائرة ال sampling) وفرق جهد مطبق على طرفها الثاني.

المُكَوِّن الثالث هو عداد رقمي تزايدي + محول رقمي - تماثلي DAC. هذان المكونان يعملان مثل الوزن المعروف مسبقاً. حيث نجد أن كلا المكونين يقومان بتوليد جهد صغير وارسالة إلى الطرف الثاني لشريحة الـ comparator. وإذا لم يتساوي كلا الجهدين يقوم العداد والـ DAC بزيادة الجهد مرة أخرى وهكذا.. وتظل هذه العملية إلى أن تقوم شريحة ال comparator بالتبليغ أن الجهد المطبق من العداد وال DAC قد تساوى أو تفوق على الجهد المطبق من عينة القياس.

عندما يحدث هذا التبليغ يتوقف العداد والـ DAC عن العمل ويتم تسجيل الرقم الذي توقف عنده. ويتم حفظ هذا الرقم في مُسجلات خاصة تسمى ADCL و ADCH ومن خلال هذا الرقم يمكننا معرفة قيمة الجهد الذي تم ادخالة لـ ADC.

9.2 تركيب الـ ADC داخل المُتحكِم ATmega16

الصورة التالية تمثل تركيب الADC للمتحكم ATmega16 (صفحة 205 دليل البيانات).

يتميز الADC الموجود داخل ATmega16 بعده امكانيات مفيدة.

أولاً: يمكن تشغيل هذا الـ ADC في وضعين. وهما 8 bit و 10 bit الاختلاف الاساسي بين الوضعين هو "حساسية القياس" للجهد التناظري وأقل قيمة جهد يمكن قياسها. سيتم تناول الدقة 8 بت فقط في هذا الفصل.

ثانياً: يتصل دخل الـ ADC بشريحة analog multiplixer والتي تتصل اطرافها بجميع أطراف البورت A. هذه الشريحة تعمل كبوابة توصيل حيث يمكن ضبطها لتوصيل أي طرف في البورت A ليعمل كدخل للـ ADC وهذا يجعل المُتحكِم ATmega16 يمتلك 7 أطراف كاملة يمكن استعمالها كدخل تماثلي ويسمى كل طرف "قناة دخل" input channel. مع مراعاة أنه يمكن تشغيل طرف واحد فقط في نفس الوقت (كما سنرى في الشرح لاحقاً).

أيضاً يحتوي الـ ADC على شريحة op-amp تعمل كمكبر جهد (يمكن تشغيله بصورة اختيارية) ودائرة مقارنة comparator تعمل على الأطراف الثلاثة الأولى PA0, PA1, PA2. (لن يتناول الكتاب شرح هذا الجزء ويمكنك الرجوع لدليل البيانات بدئاً من الصفحة 205).

المُسجِلات

يحتوي الـ ADC على مجموعة من المُسجِلات سنستخدم 3 منهم لتشغيل وضع الـ 8 بت.
ADMUX: هذا المُسجِل مسؤول عن اختيار الطرف الذي سيتم توصيله بالـ ADC لقياس الجهد كما يحتوي على مجموعة من البتات الهامة لضبط الـ analog Refrence (انظر لشرح المثال الأول).
ADCSRA: المُسجِل المسؤول عن تشغيل وايقاف الـ ADC وكذلك التحكم في سرعة تشغيله.
ADCL & ADCH: المُسجِلات المستخدمة في حفظ قيم القياس.

خطوات تشغيل الـ ADC

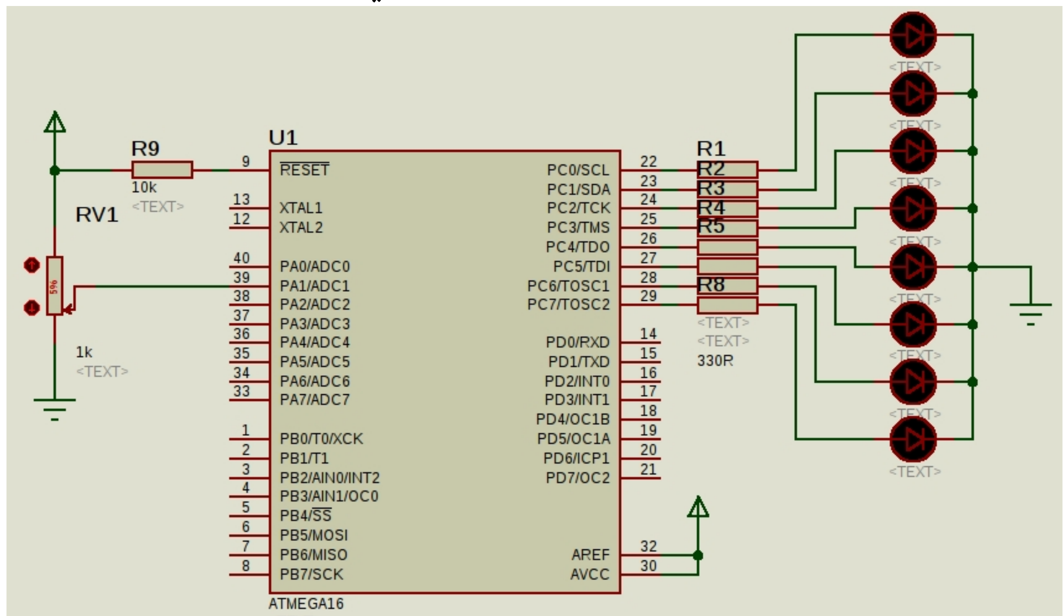
لقراءة الجهد التناظري يجب أن نقوم بمجموعة من الخطوات لتفعيل الـ ADC وضبطه بصورة صحيحة. هذه الخطوات هي كالتالي:

1. تفعيل الـ ADC (الوضع الافتراضي للـ ADC أنه غير فعال لذا سنقوم بتفعيله من المسجل ADCSRA).
2. ضبط الـ Clock والتي تتحكم في سرعة تشغيل العداد الداخلي للـ ADC والذي بدورة يتحكم في سرعة قراءة الجهد.
3. اختيار الـ channel المطلوبة (طرف القياس) وذلك من خلال ضبط الـ analog multiplixer
4. بدء عملية القياس والتحويل ثم قراءة قيمة الجهد.

9.3 المثال الأول: قراءة جهد متغير باستخدم مقاومة متغيرة

في هذا المثال سنستخدم المقاومة المتغير potentimeter كمصدر متغير للجهد. تتواجد هذه المقاومة في برنامج بروتس باسم POT-HG أو Active potentiometer ويتم توصيل الطرف العلوي لها بال vcc والطرف السفلي بال gnd أما الطرف الأوسط فسيكون مصدر الجهد المتغير.

في هذا المثال سنقوم بتوصيل طرف الجهد المتغير بالطرف PA1. كما سنستخدم 8 دايودات ضوئية لعرض قراءة ال ADC على البورت C كما هو موضح في الصورة التالية:



الكود

```
#define F_CPU 1000000UL
#include <avr/io.h>
//adc متغير لحفظ قيمة ال
volatile uint8_t adcValue;

int main(void)
{
// ضبط إعدادات البورتات //
DDRA = 0x00;
DDRC = 0xff;
```

```
// تشغيل ال adc
ADCSRA |= (1 << ADEN);

// اختيار معامل القسمة لل clock
ADCSRA |= (1 << ADPS0) | (1 << ADPS1);

// تفعيل وضع ال 8 بت //
ADMUX |= (1 << ADLAR);

// اختيار القناة (الطرف) الذي سيتم قراءة الجهد المتغير منه
ADMUX |= (1 << MUX0);

while(1)
 {
        // ابدء عملية التحويل
        ADCSRA |= (1 << ADSC);

        // انتظر حتى يتم الانتهاء من تحويل الجهد //
        while(ADCSRA & (1<<ADSC));

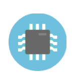
        // adcValue ضع قيمة التحويل داخل المتغير
        adcValue = ADCH;

        // C قم بعرض القيمة على البورت
        PORTC = adcValue;
 }

  return 0;
}
```

شرح الكود

في بداية البرنامج قمنا بعمل متغير 8 بت من نوع uint8_t باسم adcValue. سيستخدم هذا المتغير في حفظ القيمةالرقمية الناتجة من تحويل الجهد في ال ADC. لكن هناك كلمة غريبة ظهرت لأول مرة بجانب المتغير وهي **volatile** فما هي؟



قبل أن نتعرف على هذه الكلمة. عليك أن تتعرف على أحد الخواص الهامة للمترجمات وتدعى **Code optimaization**. كما تتذكر من الفصول السابقة إن جميع المترجمات مهمتها هي تحويل اللغات عالية المستوى (مثل السي) إلى لغة التجميع Assembly والحقيقة أن الأمر الواحد من لغة السي قد يتحول إلى 1 أو 3 أو حتى 10 أوامر من لغة التجميع. المترجمات الذكية تستيطع اختصار عدد أوامر لغة التجميع وذلك لزيادة سرعة الكود (أوامر أقل = سرعة أكبر).

هنا يأتي دور الـ code optimaization. وهي خاصية مدمجة في معظم المترجمات سواء المجانية مثل gcc أو المدفوعة مثل IAR workbench. وتهدف إلى اختصار أكبر قدر ممكن من اوامر الأسمبلي. والآن نعود للبرنامج السابق.

سنجد أن المتغير adcValue يُستخدَم لنسخ قيمة المُسجِل ADCH (وهو المُسجِل الذي يحفظ فيه القيمة الرقمية لعملية التحويل). ثم نجد أن المتغير سيستخدم في نقل هذه القمية إلى البورت C في مجموعة الأوامر التالية:

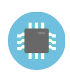
```
// adcValue ضع قيمة التحويل داخل المتغير
adcValue = ADCH;
// C قم بعرض القيمة على البورت
PORTC = adcValue;
```

في الحالة الطبيعية إذا لم نكتب كلمة volatile سنجد أن المترجم قرر حذف المتغير adcValue لأنه بلا فائدة. وذلك لأنه من الممكن أن نستبدل كلا الأمريين السابقين بأمر واحد فقط وهو وضع قيمة ADCH مباشرة داخل PORTC دون الحاجة لنسخها لمتغير adcValue.

```
PORTC = ADCH;
```

بالتأكيد يعد اختصار عدد الأوامر أمر مفيد جداً وسيجعل الكود يعمل أسرع. **لكن في هذه الحالة قد يكون كارثي. لأننا قد نحتاج هذا المتغير في اجراء عمليات حسابية أخرى** (كما سنرى في المثال القادم). وإذا قام المترجم بحذفة سيتسبب ذلك الأمر في أخطاء غير معروفة. وهنا يأتي دور كلمة volatile. والتي تخبر المترجم أن يترك المتغير adcValue ولا يقوم بحذفة.

> ملاحظة: اعتبرها قاعدة عامة، إذا كان هناك أي متغير أنت متأكد من تغير قيمته أثناء عمل المُتحكِم الدقيق ولا تريد للمترجم أن يحذفة فيجب أن تكتب كلمة volatile قبل المتغير اثناء تعريفه.



بعد الإنتهاء من تعريف المتغير. نأتي للدالة الرئيسيه main حيث بدئناً بضبط البورتات ليعمل البورت A كدخل ويعمل البورت C كخرج (لتوصيل الدايودات الضوئية الثمانية).

```
DDRA = 0x00;
DDRC = 0xff;
```

ثم تبدأ مرحلة ضبط الـ ADC عبر سلسلة من الأوامر. في البداية يجب أن نشغل الـ ADC لأن الوضع الافتراضي للـ ADC داخل جميع مُتحكِمات AVR أنه غير مُفعل لتوفير الطاقة. ويتم ذلك عبر تغيير قيمة البت ADEN وهي اختصار ADC enable. تتواجد هذه البت داخل ADCSRA

```
ADCSRA |= (1 << ADEN);
```

ADC Control and Status Register A – ADCSRA

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

المرحلة الثانية هي ضبط الـ clock الخاصة بالـ ADC. فكما تعرفنا في بداية الفصل. يحتوي الـ ADC على عداد تزايدي يُستخدَم لزيادة جهد المقارنة (الشبيه بالوزن المعياري في الميزان). ويحتاج هذا العداد الرقمي لـ Clock لتشغيله.

المشكلة لوحيدة لهذا العداد هو أنه **يجب أن يعمل ببطيء** مقارنة بسرعة المُتحكِم. فمثلاً لا يمكن أن ندخل الـ clock الرئيسية للمتحكم مباشرة إلى العداد (لأنها تكون 1 ميجا أو أعلى). لذا نستخدم شريحة تسمى بالـ Prescaler Register وهي شريحة تقوم بقسمة الـ clock الرئيسي على رقم محدد (من مضاعفات الرقم 2) مثل 2 – 4 – 8 – 16 – 32 – 64 – 128.يسمى هذا الرقم بمُعامل القسمة **Division Factor** ويتم تحديد هذا الرقم من البت ADPS0 و ADPS1 و ADPS2 الموجدتان داخل المُسجِل ADCSRA من خلال الجدول التالي

Table 85. ADC Prescaler Selections

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

بالتأكيد سيظهر سؤال هام. **على أي أساس نختار معامل القسمة وما هو تأثيره؟**
في البداية هناك قاعدة عامة لجميع الـ ADC الموجودة داخل عائلة atmega وهي أن الـ Clock الخاصة بالـ ADC يجب أن لا تزيد عن 128 كيلوهرتز (ويمكن أن تكون أقل من ذلك). فمثلاً إذا كانت الـ clock الرئيسية = 1 ميجاهرتز إذا يجب أن نختار معامل قسمة = 8 أو أكثر وذلك لأن حاصل قسمة مليون \ 8 = 125 ألف هرتز وهو رقم أقل 128 كيلو.

لنأخذ مثال آخر. لنفترض أن سرعة الـ clock الرئيسية كانت 4 ميجاهرتز. فما هو معامل القسمة المناسب؟
سنجد أن المعامل المناسب هو 32 (أو أكثر) وذلك لأن حاصل قسمة 4 مليون\32 = 125 ألف هرتز. أو يمكن استخدام معامل قسمة 64 أو 126 فكلاهما سيجعل الـ ADC clock أقل بكثير من 128 كيلوهرتز.

إذاً ما الإختلاف بين أن اختار 32 أو 64 أو 128 إذا كانت كل هذه المُعامِلات مناسبة؟

الإختلاف الأساسي هو **سرعة تحويل الجهد إلى قيمة رقمية**. فكلما زادت سرعة الـ ADC clock كلما استطاع أن يحول الجهد إلى قيمة رقمية في زمن أقل.

لنعد مرة أخرى للمثال السابق حيث كانت سرعة المُتحكِم الرئيسية 1 ميجا لذا إستخدمنا معامل قسمة = 8 وذلك عبر وضع قيمة 1 داخل كل من البت ADPS0 و ADPS1

```
ADCSRA |= (1 << ADPS0) | (1 << ADPS1);
```

الخطوة التالية هي اختيار وضع تشغيل الـ ADC (8 أو 10 بت). في حالة الـ 8 بت يتم وضع الرقم 1 داخل البت ADLAR الموجود في المُسجِل ADMUX. الوضع الإفتراضي لهذه البت = صفر (أي أي الـ ADC يعمل في وضع 10 بت).

```
// تفعيل وضع الـ 8 بت
ADMUX |= (1 << ADLAR);
```

ADC Multiplexer Selection Register – ADMUX

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

عندما يعمل الـ ADC في وضع الـ 8 بت فإن قيمة تحويل الجهد يتم تسجيلها في المُسجِل ADCH وعندما يعمل في وضع الـ 10 بت فأن قيمة التحويل يتم تسجيلها في مسجلين (لأنها

10 بت ولا يكفي مُسجِل واحد لحفظ قيمتها) وهما ADCL و ADCH ويسمى هذا الوضع بالـ left adjusted حيث توضع أول 8 بتات من النتيجة في ADCL ويوضع أخر 2 بت في ADCH

وأخيراً نقوم بضبط الطرف الذي سيكون Input channel (الطرف الذي سنقيس من خلاله الجهد الكهربي المتغير). في المثال السابق استخدمنا الطرف PA1 لذا وجب أن يتم وضع 1 داخال البت MUX0

```
// اختيار القناة (الطرف) الذي سيتم قراءة الجهد المتغير منه
ADMUX |= (1 << MUX0);
```

حيث تتحكم هذه البت مع البتات MUX(1,2,3,4) في اختيار قناة الدخل كما هو موضح في الجدول التالي.

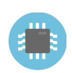
MUX4..0	Single Ended Input	Positive Differential Input	Negative Differential Input
00000	ADC0		
00001	ADC1		
00010	ADC2		
00011	ADC3	N/A	
00100	ADC4		
00101	ADC5		
00110	ADC6		

ملاحظة: الطرف ADC0 يتصل بالطرف PA0 والطرف ADC1 يتصل بالطرف PA1 وهكذا إلى نهاية البورت A

بعد الإنتهاء من ضبط الـ ADC يمكننا الآن أن نبدء عملية التحويل وقراءة الجهد. وسيتم ذلك عبر اعطاء الـ ADC الأمر ببدء عملية التحويل وذلك عبر وضع 1 داخال البت ADSC وهو اختصار لكلمة ADC start conversion

```
// ابدء عملية التحويل
ADCSRA |= (1 << ADSC);
```

هنا سيبدأ الـ ADC بقراءة الجهد وتحويله إلى قيمة رقمية. ولكن كما نعرف مسبقاً الـ ADC يعتبر بطيء جداً مقارنة بالمتحكم الدقيق لذا لا يمكننا أن نحصل على النتيجة فوراً ويجب أن



ننتظر حتى ينتهي الـ ADC من عملية التحويل. ويتم ذلك عبر الأمر

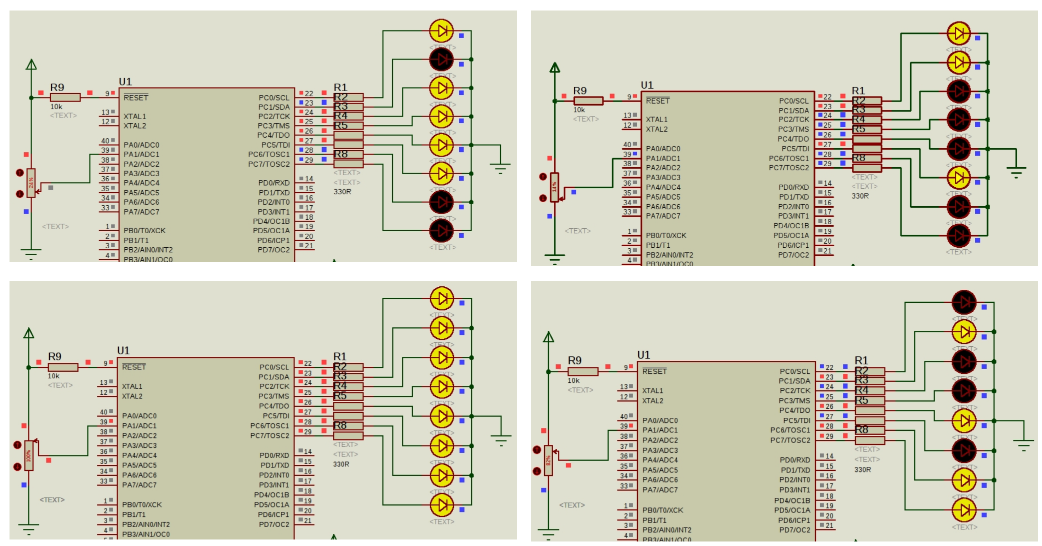
```
// انتظر حتى يتم الانتهاء من تحويل الجهد
while(ADCSRA & (1<<ADSC));
```

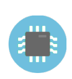
عند بدء عملية التحويل تظل البت ADSC قيمتها = 1 ولا تتحول إلى صفر إلا عند انتهاء عملية التحويل بنجاح. لذا استخدمنا الأمر السابق والذي يعني انتظر حتى تصبح البت ADSC قيمتها 0 وبذلك تتوقف الـ while.

وأخيراً يمكننا قراءة القيمة الرقمية من المُسجِل ADCH حيث نستطيع اما أن نستخدمها مباشرة أو ننسخها إلى أحد المتغيرات (في المثال السابق استخدمنا البورت C ليعرض هذه القيمة وكذلك المتغير adcValue لنسخ هذه القيمة).

```
// ضع قيمة التحويل داخل المتغير adcValue
adcValue = ADCH;
// قم بعرض القيمة على البورت C
PORTC = adcValue;
```

قم بترجمة الكود السابق ثم ابدأ محاكاة المشروع على برنامج بروتس. يمكنك أن تحرك المقاومة المتغيرة لأعلى ولأسفل لتشاهد ان الدايودات الثمانية تعرض أرقام من 00000000 إلى 11111111 كما هو موضح في الصورة التالية.





9.4 حسابات الـ ADC

في المثال السابق استطعنا أن نحول الجهد التناظري المتغير إلى أرقام بين 00000000 إلى 11111111 (ما بين 0 إلى 255 بالصيغة العشرية). لكن هذه الأرقام لا تمثل قيمة فرق الجهد بصورة مباشرة لذا سنتعرف على في هذا الجزء على بعض المعادلات البسيطة التي ستساعدنا على حساب فرق الجهد وكذلك حساب قيم الحساسات التماثلية.

تحويل القيمة الرقمية إلى فرق جهد

المعادلة التالية تقوم بتحويل القيمة الرقمية إلى فارق جهد.

$$Voltage = \frac{DigitalValue * Vref}{2^n}$$

- Digital Value: القيمة الرقمية الموجودة في ADCH (بالصيغة العشرية)
- Vref: الجهد المرجعي للـ ADC (أنظر للشرح بالأسفل)
- n: وضع دقة التشغيل (8 بت أو 10 بت)

الـ Vref هو الجهد المرجعي الذي يعتمد عليه الـ ADC لحساب الجهد المتغير. هذا الجهد يساهم في تحديد حساسية قياس الـ ADC (كما سنرى في الجزء التالي) ويتم تحديدة من خلال البتات REFS0 و RFS1. في المثال السابق تركنا هذه البتات على الوضع الافتراض (صفر) والذي يعني أن الجهد المرجعي Vref = Vcc = 5 volt.

الصورة التالية توضح طريقة تغير الـ Vref بالتغيير قيمة RFS1 , RFS0 في المُسجِل ADMUX

- **Bit 7:6 – REFS1:0: Reference Selection Bits**

These bits select the voltage reference for the ADC, as shown in Table 83. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set). The internal voltage reference options may not be used if an external reference voltage is being applied to the AREF pin.

Table 83. Voltage Reference Selections for ADC

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

لنفترض أن القيمة الرقمية التي حصلنا عليها = 128 وكان الـ ADC يعمل على الوضع 8 بت.
إذا بالتعويض في المعادلة السابقة نجد أن قيمة الجهد الذي تم قياسه يساوي

$$\frac{128*5}{2^8}=\frac{128*5}{256}=2.5\,volts$$

مثال أخر: لنفترض أن القيمة الرقمية التي حصلنا عليها = 56. إذا بالتعويض في المعادلة السابقة نجد أن قيمة فرق الجهد هي

$$\frac{56*5}{2^8}=\frac{56*5}{256}=1.09\,volts$$

حساسية القياس

تعرف حساسية القياس بأنها أقل جهد يمكن للـ ADC أن يقيسه ويتعرف عليه بصورة صحيحة. وتعتمد حساسية القياس لا ADC على عاملين وهما وضع التشغيل (8 أو 10 بت) و الجهد المرجعي Vref. كلا العاملين يؤثران بشكل كبير جداً في دقة القياس. المعادلة التالية تمثل حساسية القياس.

$$sensitivity=\frac{Vref}{2^n}$$

مجدداً تمثل n وضع القياس (8 أو 10 بت). فمثلاً لو أن vref = vcc = 5 volt ويعمل الـ adc بدقة 8 بت فهذا يعني أن حساسية القياس تساوي

$$\frac{5}{2^8}=\frac{5}{256}=0.019\,volt$$

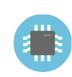
مما يعني أن أقل جهد يمكن قياسه هو 0.019 فولت (نحو 20 مللي فولت).

> يتحكم الـ Vref في أقصى جهد يمكن قياسه فمثلا لو كان Vref = 3 فولت إذا أقصى جهد يمكن لـ ADC أن يقيسه هو 3 فولت

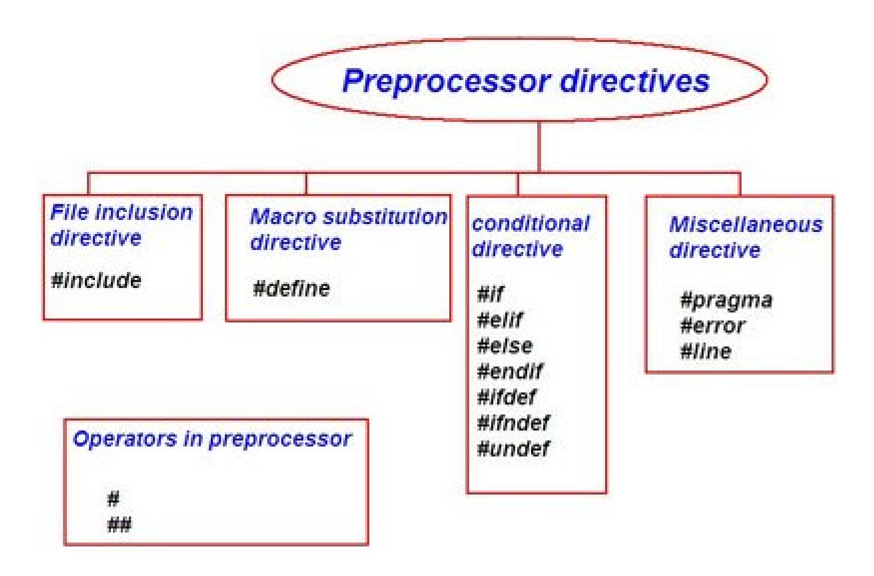
الفصل العاشر

" أحدهم يجلس في الظل اليوم، لأن شخصا آخر قام بزرع شجرة في ذاك المكان منذ زمن بعيد "

وارين بافيت - رجل أعمال ومستثمر أمريكي



.10 المعالج التمهيدي وصناعة المكتبات البرمجية



في هذا الفصل سنتحدث عن أكواد C preprocessor حيث سنتعرف على الفارق بين الأوامر التنفيذية والأوامر التوجيهية وأهميتها بصورة مفصلة مثل الأمر #include وكذلك define وكذلك سنتعرف على كيفية صناعة المكتبات البرمجية libraries. مع شرح مثال لعمل uart driver على صورة مكتبة.

- ✔ الأوامر التنفيذية والأوامر التوجيهية
- ✔ بعض استخدامات الـ C – Preprocessor
- ✔ قواعد صياغة الأوامر التوجيهية
- ✔ طرق كتابة Function-like macros
- ✔ تصميم المكتبات البرمجية
- ✔ مثال: تصميم مكتبة UART driver

10.1 الأوامر التنفيذية والأوامر التوجيهية

الأوامر التنفيذية هي أي أمر مباشر في لغة السي مثل for – while – if وجميع الأوامر الرياضية أو المنطقية AND – OR – NOT – XOR، كل هذه الأوامر تستخدم "لتنفيذ" أمر مطلوب. على العكس **الأوامر التوجيهية** Directive (تسمى أيضاً أوامر إرشادية) هي أوامر لا تدخل مباشرة في تركيب الكود ولكن تستخدم في إرشاد المترجم compiler لأداء بعض الأمور.

المترجم GCC يعرف مسبقاً جميع أوامر لغة السي المعيارية لكنه لا يعرف الدالة delay_ms والتي لا تتواجد إلا في المكتبة البرمجية delay.h لذا نستخدم الأمر التوجيهي

```
#include <delay.h>
```

وذلك لنرشد المترجم للمكان الذي يحتوي على دوال التأخير الزمني التي سنحتاجها في البرامج وذلك حتى يستطيع المترجم أن يصل إلى هذه الدوال بصورة صحيحة ويقوم بتحويلها إلى صيغة الهيكس.

بعض استخدامات C - preprocessor

الأوامر التوجيهية ليست مقتصرة فقط على الأمر include وإنما هناك العديد من الأوامر منها:

- **include –** أمر لتضمين ملف أو مكتبة معينة.
- **define –** أمر لتعريف دلالات لكلمات كأن نعرف أنه كلما وردت كلمة Pi في الكود فهو يعني 3.14 وكذلك لإعطاء معرّف(رمز) لكتلة أسطر برمجية مع إمكانية وجود وسطاء arguments وهذا ما يسمى function-like macros.
- **pragma –** أمر لتحديد بعض الأوامر للمترجم compiler
- **الترجمة الشرطية conditional compilation –** مجموعة أوامر تستخدم في الترجمة حسب شروط معينة، مثال: نخبر المترجم أنه لو كنت في نظام تشغيل ويندوز قم بتضمين المكتبة الفلانية أما لو كنت في نظام تشغيل لينكس فقم بتضمين مكتبة أخرى.

10.2 قواعد الأوامر التوجيهية C - preprocessor syntax

أي سطر برمجي يبدأ برمز المربع # hash فإن ما يليه هو أمر سيوجه إلى ال preprocessor .

#include

يعتبر الأمر أكثر الاستخدامات شيوعاً من أوامر الـ C - preprocessor وهو أمر توجيهي من أجل تضمين مكتبة (وهي عبارة عن مجموعة تعريفات) أو حتى ملفات مصدرية أخرى.

مثال:

بفرض الكود المصدري للملف main.c هو التالي:

```
#include <avr/delay.h>
#include "device_conf.h"

int main(void)
{
set_port_output;
while(1)
  {
        PORT_on;
        _delay_ms(1000);
        PORT_off;
        _delay_ms(1000);
  }
return 0;
}
```

نلاحظ أن التضمين الأول كان بين قوسين <> والثاني بين إشارتين "" ،حيث أن الاستخدام الأول يكون عندما نريد أن يتم البحث عن الملف المحدد ضمن المسارات المتاحة في إعدادات المترجم. والاستخدام الثاني يكون عندما نريد البحث عن الملف المحدد ضمن المجلد المصدري نفسه.

أيضاً نلاحظ وجود بعض الأوامر الغريبة مثل PORT_on و الأمر PORT_of. الحقيقة أن تفسير هذه الأوامر يقع في الملف **device_conf** والذي يحتوي على الكود التالي:

```
#include <avr/io.h>
#define PORT_on PORTD=0xFF
#define PORT_off PORTD=0x00
#define set_port_output DDRD=0xFF
```

عند القيام بعملية الترجمة Compiling ما سيحدث أن المعالج التمهيدي للغة السي سيبدأ معالجة الكود قبل عملية الترجمة الحقيقة (تحويل الملف إلى هيكس) وسيتم معالجة الأمر **include** ليصبح البرنامج السابق كالتالي:

(للتبسيط لن نذكر نتائج تضمين مكتبة avr/delay.h أو مكتبة avr/io.h):

```
#define PORT_on   PORTD=0xFF
#define  PORT_off  PORTD=0x00
#define set_port_output  DDRD=0xFF

int main(void)
{
set_port_output;
while(1)
  {
        PORT_on;
        _delay_ms(1000);
        PORT_off;
        _delay_ms(1000);
   }
return 0;
}
```

نتيجة تنفيذ الأمر **include**

#define

يُستخدَم هذا الأمر التوجيهي directive لتعريف كلمات كرموز (كلمات مفتاحية) ترد في الكود ليتم استبدالها بالقيمة المحددة (أرقام أو حروف أو أسطر برمجية أخرى) مع أننا سنرى لاحقاً أن ال preprocessor لا يستطيع التمييز بينها.

ففي الكود السابق سيقوم ال preprocessor بالبحث عن أماكن ورود PORT_on و PORT_off

و set_port_output ويقوم باستبدالها بما هو مذكور في تعريفها. أي أن الكود سيصبح كالتالي (بعد أن ينتهي المعالج من معالجة الأمر define):

```
#define PORT_on  PORTD=0xFF
#define  PORT_off  PORTD=0x00
#define set_port_output  DDRD=0xFF

int main(void)
{
set_port_output;
while(1)
  {
        PORTD=0xFF;
        _delay_ms(1000);
        PORTD=0x00;
        _delay_ms(1000);
  }
return 0;
}
```


ملاحظة مهمة: إن ما يرد بعد الأمر التوجيهي لا يتم معالجته أو تنقيح أخطائه، مثلاً لو قمنا بالتعديل التالي وهو إضافة تعليق

```
#define PORT_on  PORTD=0xFF
#define  PORT_off  PORTD=0x00          //set portD off
#define set_port_output  DDRD=0xFF
```

سيصبح الكود الرئيسي بعد الـ Preprocessor كالتالي:

```
while(1)
  {
        PORTD=0xFF;
        _delay_ms(1000);
        PORTD=0x00          //set portD off  ;
        _delay_ms(1000);
  }
```


نلاحظ أن الكود أصبح يحتوي على خطأ حيث أصبحت الفاصلة المنقوطة بعد التعليق، فال preprocessor ليس من مهمته فهم ما بعد الأوامر التوجيهية وإنما فقط استبدال الرموز بقيمتها.

10.3 function-like macros

يمكننا من خلال ال C preprocessor إنجاز ما **يشبه الدوال** ولذلك تسمى **function-like** مع اختلاف جوهري بين الدوال و function-like macros، شكل الشبَه الوحيد هو إمكانية إنجاز macros يمكنه أن يستقبل بعض المتغيرات، أما من الناحية التنفيذية فليس هناك أي وجه شبه:

- **function –** كتلة من الكود البرمجي يمكن أن نستدعيها بأي مكان من البرنامج الأساسي لتقوم بالتنفيذ واستخدام المتغيرات في حال وجودها مع إمكانية إرجاع قيمة بعد التنفيذ.
- **Function-like macros –** مجموعة من الأسطر البرمجية المختصرة برمز، و عند إيراد هذا الرمز في الكود فهو بمثابة إيراد هذه الأسطر البرمجية كما هي، بخلاف مبدأ الدوال المعتمد على الاستدعاء مع وجود الكود دون تكرار.

لابد من التنويه إلى الجانب السلبي لاستخدام ال C preprocessor وهو حجم البرنامج النهائي، إذا أن استخدام ال C preprocessor لإنجاز ما يشبه الدوال function-like macros يؤدي إلى تضاعف حجم البرنامج، وذلك لأن استخدام ال C preprocessor لا يتعدى في النهاية عن اختصار أسطر برمجية بكلمات مفتاحية.

10.4 قواعد كتابة الماكرو macros syntax

لصناعة macro يمكنه استقبال متغيرات يجب أن نتبع اسم المايكرو (الرمز) بأقواس تحوي أسماء المتغيرات مباشرة ، دون تحديد نوعها كما في الدوال، إذ لا يهم ال function-like macros نوع المتغيرات التي يتم تمريرها إليها.

المجموعة التالية من الأكواد تمثل أشهر ال macros التي يتم استخدامها عادة لتسهيل برمجة المُتحكِمات واختصار الكثير من الوقت. مع العلم أن هذه الـ macros يمكن استخدامها مع أنواع مختلفة من المُتحكِمات الدقيقة عبر أي مترجم يدعم لغة السي المعيارية فيمكنك مثلاً أن تستخدمها في برمجة مُتحكِمات ARM cortex.

```
#define BIT_SET(ADDRESS,BIT) (ADDRESS |= (1<<BIT))
#define BIT_CLEAR(ADDRESS,BIT) (ADDRESS &= ~(1<<BIT))
#define BIT_CHECK(ADDRESS,BIT) (ADDRESS & (1<<BIT))
#define BIT_FLIP(ADDRESS,BIT) (ADDRESS ^= (1<<BIT))
```

لنأخذ أحد ال function-like macros كمثال

```
#define BIT_SET(ADDRESS,BIT) (ADDRESS |= (1<<BIT))
```

اسم macro هو BIT_SET والمتغيرات التي يستقبلها هي ADDRESS, BIT حيث يمكننا أن نستخدمه داخل البرنامج الرئيسي كالتالي:

```
BIT_SET(PORTB,5);
```

وهذا مكافئ تماماً للأمر الذي يقوم بوضع 1 داخل أي بت من أي مُسجِل مطلوب كالتالي:

```
PORTB |= (1<<5);
```

أما الماكرو **BIT_CLEAR** فهو مكافئ للأمر الذي يضع 0 داخل أي بت من أي مُسجِل كالتالي:

```
PORTB &= ~(1<<5);
```

10.5 مراجع إضافية

سلسلة مقالات ال C – preprocessor العربية (المصدر الأصلي للجزء المشروح بالأعلى)

http://www.atadiat.com/c-preprocessor-part1
http://www.atadiat.com/c-preprocessor-part-2
http://www.atadiat.com/c-preprocessor-part-3

مراجع أجنبية إضافية

http://www.mybitbox.com/2012/12/robust-c-code-part-3-wrapping-c/

http://www.mybitbox.com/2012/12/robust-c-code-part-2-advanced-c-preprecessor/

http://www.mybitbox.com/2012/11/robust-c-code-part-1-c-preprocessor/

http://www.cprogramming.com/tutorial/cpreprocessor.html

http://www2.hh.se/staff/vero/embeddedProgramming/lectures/printL2.pdf

http://www.phaedsys.com/principals/bytecraft/bytecraftdata/bcfirststeps.pdf

http://en.wikipedia.org/wiki/C_preprocessor

10.6 تصميم المكتبات البرمجية في لغة السي

المكتبات البرمجية تعد من أفضل أساليب "تجزئة الكود" فهي تسمح للمبرمجين بصناعة نماذج من الأكواد البرمجية التي يمكن إعادة استخدامها بسهولة في أكثر من تطبيق. مثلاً لنفترض أنك تريد تشغيل شاشة أو محرك أو حساس خاص مع المُتحكِم الدقيق، يمكنك أن تكتب الأكواد الخاصة بتشغيله (أو كما تسمى driver code) داخل البرنامج الرئيسي وينتهي الأمر عند هذا الحد. لكن عندما تريد أن تستخدم نفس الحساس في مشروع آخر سيتوجب عليك أن تعيد كتابة هذا الكود مرة أخرى.

المكتبات البرمجية توفر الوقت والمجهود لإعادة كتابة هذه الأوامر فكل ما عليك فعله هو أن تصمم الـ driver code مرة واحدة وتضعها في library ثم تستخدم هذه المكتبة في أي مشروع ترغب به.

> ملاحظة: تسمى المكتبات التي تشغل أي جزء داخل المُتحكِم الدقيق مثل ADC , UART, i2C, GPIO أو أي مكون إلكتروني خارجي Software driver.

أيضاً تساعد المكتبات على تسهيل العمل الجماعي بين الأفراد. فمن الصعب على مجموعة مكونه من 10 مبرمجين مثلاً أن يكتبوا كل الأكواد داخل ملف واحد فقط. وبدل من ذلك يتم تقسيم الأكواد الكبيرة إلى مجموعة من الـ Modules (الوحدات) والتي عادة تكون مجموعة من المكتبات البرمجية ويتولى كل شخص من فريق التطوير العمل على أحد هذه الوحدات بالتوازي مع باقي الفريق.

تركيب المكتبات في لغة السي

تتكون المكتبات في لغة السي عادة من بناء نوعين من الملفات، الأول يسمى Header file أو كما يحب البعض أن يسميه Definition file (ملف التعريفات) ويكون على هيئة الامتداد **file.h** والثاني هو الملف التطبيقي Implementation file والذي يحتوي على الأوامر الحقيقية للمكتبة ويكون من نوع ملفات السي **file.c**

مثلاً لنفترض أننا نريد تصميم مكتبة لتشغيل الاتصال التسلسلي UART وذلك لتسهيل استخدام UART دون الحاجة لإعادة كتابة جميع الدوال في كل برنامج.

10.7 خطوات صناعة المكتبة

تمر أي مكتبة برمجية بمجموعة من الخطوات حتى نضمن أن نحصل على كود فعال ويعمل بصورة جيدة. ويستحسن أن تتبع هذه الخطوات في أي من المكتبات التي ستقوم بكتابتها مستقبلاً.

الخطوة الأولى: فهم الكود والتجربة الأولية

في البداية عليك أن تفهم العنصر أو الجهاز الذي تريد أن تكتب له هذا الـ driver وذلك عبر اختبار بعض الأمثلة في البرنامج الأساسي أولاً ثم تحويل هذه الأمثلة لمكتبة. بما أننا سنكتب UART driver فهذه الخطوة قد تمت بالفعل في الفصل الخاص بالـ UART حيث يمكنك الرجوع إليه مرة أخرى لمراجعة الأوامر والدوال التي سنستخدمها في الخطوات التالية.

الخطوة الثانية: تكوين ملف الهيدر uart.h

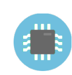
بعد القيام بجميع التجارب التي تختبر الأكواد التي سنحتاجها سنقوم بعمل ملف الهيدر Header File والذي يتكون من مجموعة من أوامر الـ C – preprocessor مضاف إليها جميع أسماء الثوابت والمتغيرات وكذلك الـ function prototypes (أسماء الدوال). الكود التالي يمثل الهيكل الأساسي لأي ملف هيدر (سواء للغة السي أو السي++). ويمكنك عمل هذا الملف باستخدام أي محرر نصوص مثل notepad++

```
#ifndef  __LIBRARYNAME_H__
#define  __LIBRARYNAME_H__

        هنا تكتب جميع التعريفات للدوال والمتغيرات المختلفة

#endif
```

يتم استبدال LIBRARYNAME باسم المكتبة المطلوب مع مراعاة أن تكون جميع الأحرف من نوع upper-case (حروف كابيتال) فمثلاً لو كان اسم المكتبة **uart** سنكتب داخل ملف الهيدر الصيغة التالية (أيضاً لا تنسى كتابة _H__ بعد اسم المكتبة):



```
#ifndef  __UART_H__
#define  __UART_H__

        .........

#endif
```

لنأخذ مثال آخر. لنفترض أن اسم المكتبة هو **uartDriver** فسيكون شكل ملف الهيدر كالتالي:

```
#ifndef  __UARTDRIVER_H__
#define  __UARTDRIVER_H__

        .........

#endif
```

الآن يمكننا أن نكتب جميع التعريفات المطلوبة والتي تشمل أسماء المتغيرات والثوابت وكذلك أسماء الدوال التي سنستخدمها وأي C – preprocessor قد نحتاجها لتشغيل أوامر المكتبة. إذا كنت تتذكر من الفصل الخاص بالـ UART سنجد أنها هناك مجموعة متغيرات ودوال أسماؤها كالتالي:

المتغيرات

```
uint16_t UBRR_Value          // المتغير المسئول عن تحديد سرعة السيريال
```

الدوال

```
void UART_init()                  // تشغيل وضبط UART
void UART_send_char(char data)    // إرسال حرف
char UART_receive_char()          // استقبال حرف
void UART_send_string(char *data) // إرسال كلمة أو جملة نصية
```

الآن كل ما عليك فعله هو إضافة كل التعريفات السابقة إلى ملف الهيدر مع إضافة علامة الفاصلة المنقوطة **(;)** بعد كل تعريف (سواء كان متغير أو دالة). ليصبح شكل ملف الهيدر:

```
#ifndef  __UARTDRIVER_H__
#define  __UARTDRIVER_H__

uint16_t UBRR_Value;
void UART_init(uint16_t baud_rate);
void UART_send_char(char data);
char UART_receive_char();
void UART_send_string(char *data);

#endif
```

تتبقى خطوة أخيرة للانتهاء من الملف وهي إضافة المكتبات التي قد تحتاجها الدوال السابقة. إذا كنت تذكر الدالة UART_init() كانت تستخدم المكتبة البرمجية math.h لحساب الرقم المسؤول عن تحديد سرعة الـ UART وكذلك تستخدم المكتبة avr/io.h في ضبط قيم المُسجِلات لذا سنضيف كلا المكتبتين إلى ملف الهيدر. أيضاً نحتاج أن نكتب تعريف سرعة المُتحكِم لأننا سنحتاج **F_CPU** داخل الدالة **UART_init**

كما يستحسن أن نضيف بعض التعليقات في بداية المكتبة لتوضح تاريخ صناعة هذه المكتبة والوظائف التي تحققها. الشكل النهائي يمثل الصورة النهائية من ملف الهيدر.

```
/*        UART driver  v.0.1
          Date:   9-2015
 This library designed to make a simple and re useable UART driver for both
ATmega16 and ATmega32
*/

#ifndef   __UARTDRIVER_H__
#define   __UARTDRIVER_H__

#define F_CPU 16000000UL
#include <avr/io.h>
#include <math.h>

uint16_t UBRR_Value;    // Variable to store the calculations needed to set speed

void UART_init(uint16_t baud_rate);           // Initiate the UART
void UART_send_char(char data);       // Send singe character
char UART_receive_char();             //  Receive single character
void UART_send_string(char *data);     // Send array of characters

#endif
```

ملاحظة: الملف **uart.h** و **uart.c** مرفقان مع الكتاب لتسهيل نسخ النصوص أو تعديلها

الخطوة الثالثة: تكوين ملف التطبيق uart.c

الخطوة الأخيرة هي كتابة الأوامر الحقيقية لجميع الدوال داخل ملف جديد بنفس اسم المكتبة ولكن بامتداد .c مثل أن نكتب **uart.c** مع ملاحظة أنه لا يتم تعريف أي متغيرات جديدة داخل هذا الملف (لأننا قد كتبنا تعريفات المتغيرات في ملف الهيدر).

كما يجب أن يحتوي ملف التطبيق على أمر التضمين include وذلك لإضافة ملف الهيدر إلى ملف التطبيق كما هو موضح في الكود التالي (والذي يمثل محتوى الملف **uart.c**).

```
#include "uart.h"

void UART_init(uint16_t baud_rate)
{
  uint16_t UBRR_Value = lrint ( (F_CPU / (16L * baud_rate) ) -1);
  UBRRL = (uint8_t) UBRR_Value;
  UBRRH = (uint8_t) (UBRR_Value >> 8);
  UCSRB = (1<<RXEN) | (1<<TXEN);
  UCSRC  |= (3<<UCSZ0);
}

void UART_send_char(char data)
{
   while( ! (UCSRA & (1<<UDRE) ) );
     UDR = data;
}

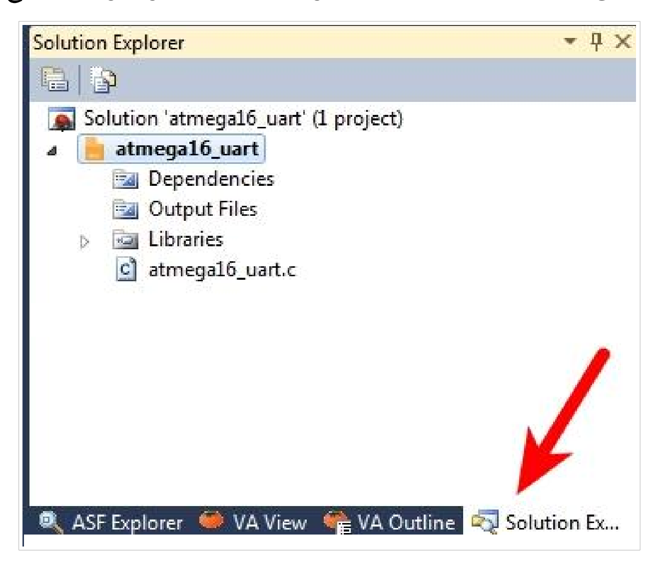
char UART_receive_char()
{
   while (! (UCSRA & (1 << RXC) ) );
         return UDR;
}

void UART_send_string(char *data)
{
    while(*data > 0)
          UART_send_char(*data++);
    UART_send_char('\0');
}
```

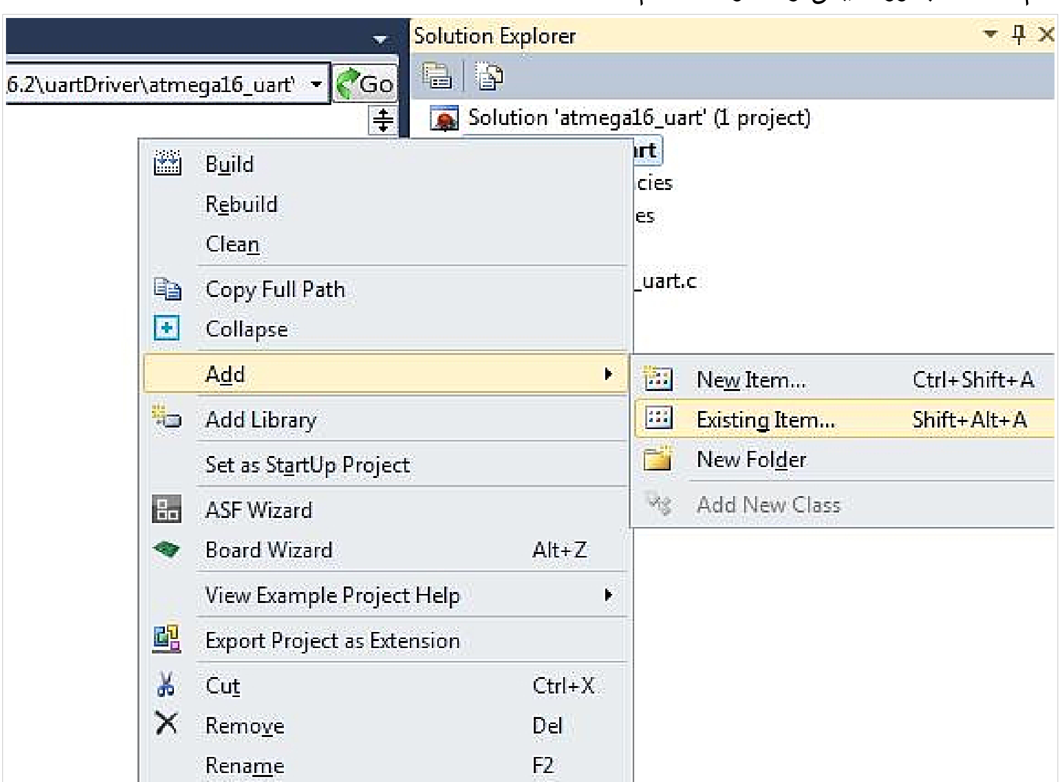
10.8 تجربة المكتبة في برنامج ATmel studio

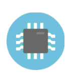
والآن بعد أن انتهينا من كتابة الـ UART driver لنقم باختباره. في البداية سنقوم بإضافة ملفات المكتبة إلى مشروع جديد داخل برنامج ATmel studio ويمكنك ذلك بطريقتين، الأولى أن تنسخ ملف uart.c و uart.h إلى نفس مجلد المشروع C:/documents/atmel أو أن تقوم بذلك بسهولة من متصفح ملفات المشروع من داخل البرنامج نفسه.

في البداية توجه إلى **Solution Explorer** من الجانب الأيمن من البرنامج

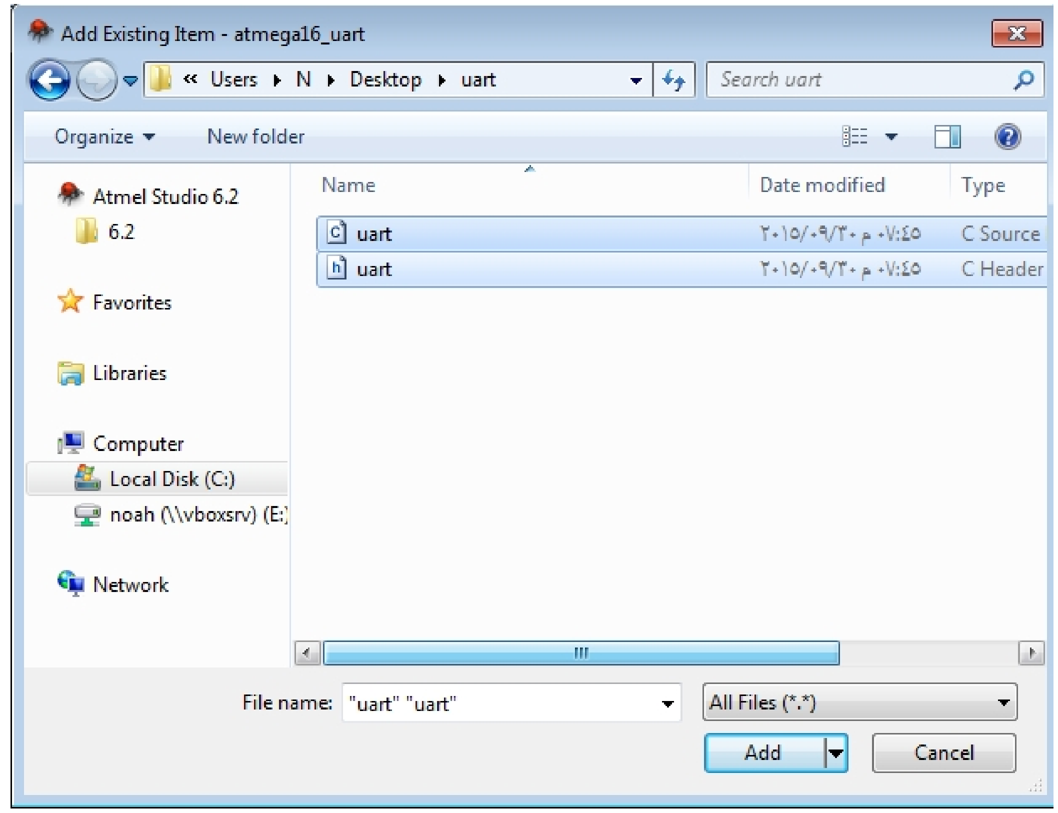


ثم اضغط بالزر الأيمن وأختر **Add** ثم **Existing item**

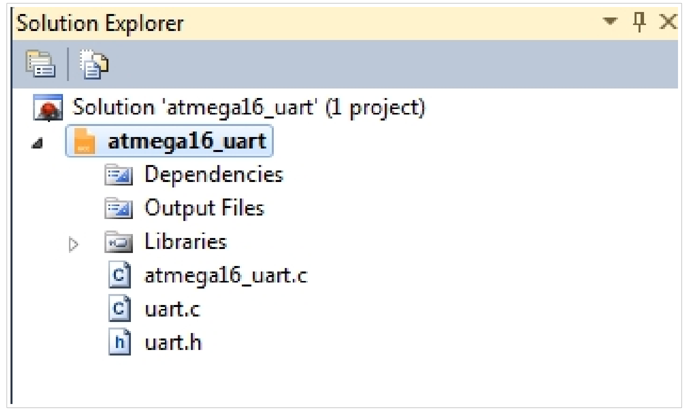




والآن اختر الملفين **uart.c** و **uart.h**



بعد إضافة الملفات ستلاحظ ظهورها في قائمة الـ **Solution Explorer**



والآن لنكتب برنامج بسيط لإرسال حرف H باستخدام الـ **uart driver**. في البداية سنقوم بعمل include للملف uart.h داخل البرنامج الأساسي (لاحظ أنه لا داعي لإعادة كتابة **define** **F_CPU** ولا داعي لإضافة المكتبة **avr/io.h** لأن كلاهما مضاف بالفعل داخل الملف **uart.h**.

```
#include "uart.h"
#include <util/delay.h>

int main(void)
{
	UART_init(9600);

	while(1)
	{
		UART_send_char('H');
		_delay_ms(1000);
	}

return 0;
}
```

صورة الكود الذي سيختبر المكتبة داخل برنامج ATmel studio

uart.h uart.c atmega16_uart.c ASF Wizard

atmega16_uart.c C:\Users\N\Documents\Atmel Studio\6.2\uartDriver\atmega16_uart\ Go

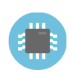
```
#include "uart.h"
#include <util/delay.h>

int main(void)
{
    UART_init(9600);

    while(1)
    {
        UART_send_char('H');
        _delay_ms(1000);
    }
}
```

لاحظ أن أمر تضمين المكتبة uart.h يكون مكتوب بين علامتي "" وذلك لأن ملف المكتبة يكون موجود **داخل نفس مجلد المشروع** بينما المكتبات الأخرى مثل delay.h تكتب بين علامتي <> لأنها تكون **موجودة داخل مجلد المكتبات العام** التابع لل toolchain.

الصور التالية تمثل شكل الملف uart.h و uart.c داخل برنامج ATmel Studio.



```
uart.h  uart.c  atmega16_uart.c  ASF Wizard
uart.h    C:\Users\N\Documents\Atmel Studio\6.2\uartDriver\atmega16_uart'  Go

/*          UART driver  v.0.1
            Date:   9-2015
     This library designed to make a simple and re usable UART driver
     for both atmega16 and atmega32
*/

#ifndef    __UARTDRIVER_H__
#define   __UARTDRIVER_H__

#define F_CPU 16000000UL
#include <avr/io.h>
#include <math.h>

// Variable to store the calculations needed to set speed
uint16_t UBRR_Value;

void UART_init(uint16_t speed);      // Initiate the UART
void UART_send_char(char data);       // Send singe character
char UART_receive_char();           //  Receive single character
void UART_send_string(char *data);     // Send array of characters

#endif
```

الملف **uart.c**

```
uart.h  uart.c  atmega16_uart.c  ASF Wizard
UART_init.UBRR_Value    uint16_t UBRR_Value = lrint ( (F_CPU / (16L * baud_rate) ) -1)  Go

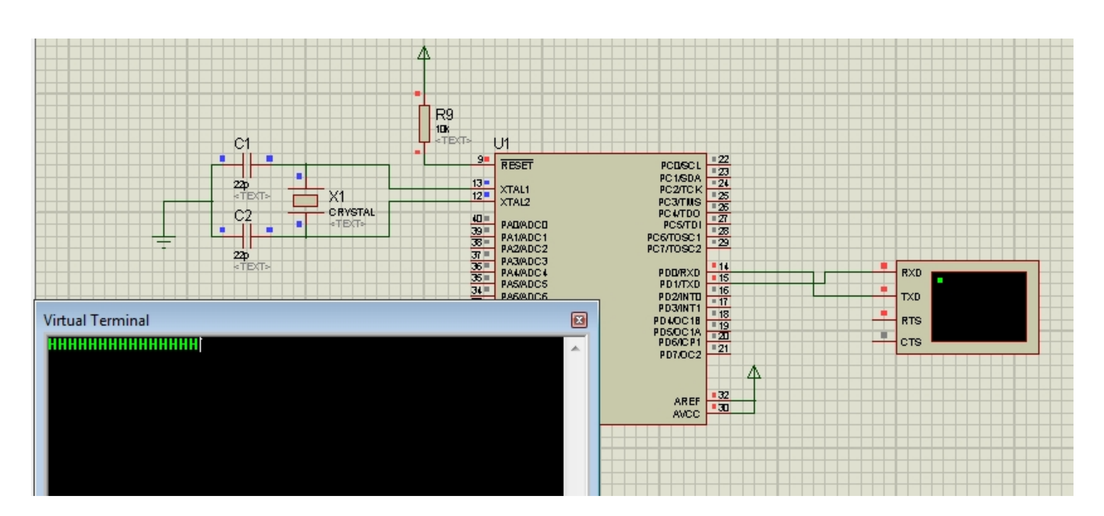
#include "uart.h"

void UART_init(uint16_t baud_rate)
{
    uint16_t UBRR_Value = lrint ( (F_CPU / (16L * baud_rate) ) -1);
    UBRRL = (uint8_t) UBRR_Value;
    UBRRH = (uint8_t) (UBRR_Value >> 8);
    UCSRB = (1<<RXEN) | (1<<TXEN);
    UCSRC |= (3<<UCSZ0);
}

void UART_send_char(char data)
{
     while( ! (UCSRA & (1<<UDRE) ) );
        UDR = data;
}

char UART_receive_char()
{
     while (! (UCSRA & (1 << RXC) ) );
     return UDR;
}
```

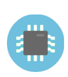
بعد الانتهاء من كتابة الكود قم بترجمته واستخلاص ملف الهيكس ثم قم بمحاكاة البرنامج بنفس الدائرة المذكورة في المثال الأول في الفصل الخاص بال UART.والصورة التالية تمثل نتيجة المحاكاة.



الفصل الحادي عشر

" المسافة بين أن تكون شخصا عادياً وبين أن تكون عبقريا هي مدى النجاح الذي تحققه "

بروس فيرشتاين - مؤلف وكاتب أمريكي

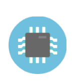


11. أنظمة الوقت الحقيقي RTOS



يشرح هذا الفصل استخدام أنظمة تشغيل الوقت الحقيقي Real Time OS لتشغيل المهام المتعددة Multitasking وأنظمة الاستجابة السريعة. حيث سيتم تناول نظام FreeRTOS في هذا الفصل باعتباره أفضل نظام RTOS مجاني (ومفتوح المصدر).

- ✔ مقدمة عن مفهوم الـ RTOS
- ✔ كيف يتم تنفيذ أكثر من مهمة
- ✔ كيف تعمل نواة النظام FreeRTOS Kernel
- ✔ تشغيل FreeRTOS على أي مُتحكِم System Porting to any AVR
- ✔ إعدادات النظام
- ✔ مثال: تشغيل 3 مهمات مختلفة
- ✔ شرح طرق إدارة المهام



11.1 مقدمة عن أنظمة الوقت الحقيقي Real Time Systems

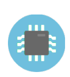


تعرف أنظمة الوقت الحقيقي بأنها أنظمة الحوسبة أو الأنظمة المدمجة التي تستخدم في أداء مجموعة من المهمات تتطلب كل مهمة استجابة زمنية وسرعة تنفيذ محددة، وتنقسم هذه الأنظمة إلى 3 أنواع:

Soft Real Time Systems وهي الأنظمة التي تسمح ببعض التأخير المحدود في التنفيذ أو الإستجابة للمهمات المطلوب تنفيذها. أمثلة على ذلك، الهواتف النقالة، مشغلات الألعاب، أجهزة عرض الفيديو. كل ما سبق أنظمة يمكن التغاضي عن حدوث تأخير زمني بسيط أثناء تشغيلها. فمثلاً لن تحدث مشكلة إذا تأخرت شاشة الهاتف النقال بضعة مللي ثانية في الاستجابة لأوامر المستخدم.

Hard Real Time Systems وهي الأنظمة التي **دائماً ما يجب أن تنجح** في تنفيذ جميع المهمات الخاصة بها في الزمن المطلوب وبدون أي تاخير في الإستجابة مهما كان. وحدوث أي تأخير يعني فشل النظام ككل وقد يؤدي إلى كارثة. من الأمثلة على هذه الأنظمة وحدات التحكم في المفاعلات النووية، الصواريخ، الطائرات الحربية، بعض الأجهزة الطبية مثل أجهزة تنظيم ضربات القلب.

Firm Real Time Systems تعتبر مماثلة لـ Hard Real Time systems من ناحية سرعة الإستجابة والتنفيذ لكن الإختلاف هو أنه في حالة الفشل **بضع مرات** للاستجابة المطلوبة فلن يؤدي ذلك إلى فشل النظام ككل. ومثال على ذلك جهاز الراوتر اللاسلكي Wireless Router (الذي يبث الإنترنت بصورة لاسلكية للهواتف أو الأجهزة المحمولة) يحتوي على شريحة تعمل على إستقبال وإرسال حزم البيانات Data packets باستجابة عالية ومثالية لكن إذا حدث أن فقدت بعض الحزم أو لم يستطع الراوتر التعامل معها فإن **ذلك لا يعني فشل للنظام ككل فهذا يعتبر خطأ مقبول طالماً أنه ليس متكرر بكثرة.**



11.2 طرق تصميم الـ Real Time Embedded systems

هناك العديد من الطرق لتصميم نظم مدمجة عالية الإستجابة بعضها يتسم بالبساطة والبعض الآخر مُصمم ليعالج الوظائف المعقدة والمتداخلة. من هذه الطرق هناك نوعين أساسين:

الطريقة الأولى: كتابة الوظائف التي لا تحتاج استجابة سريعة داخل الدالة الرئيسية Main function بينما المهمات التي تتطلب استجابة لحدث خارجي معين أو تحدث بصورة دورية يتم وضعها داخل دوال مقاطعات interrupts.

```
#include "library1"
#include "library2"
int main()
{
 some_data

  while(1)
    {
         // هنا تكتب المهمات ذات الأهمية التقليدية
    }
 return 0;
}
```

```
ISR(type_of_ISR)
{
 // كود معالجة سريع لأحد المهمات عند حدوث مقاطعة
}
```

```
ISR(type_of_ISR)
{
  // كود معالجة سريع لأحد المهمات عند حدوث مقاطعة
}
```

يعيب هذه الطريقة أنها لا تصلح إلا لعدد محدود جداً من المهمات كما أن كتابة الكود البرمجي لأكثر من مهمة داخل الدالة الرئيسية يجعل الكود معقد.

الطريقة الثانية: استخدام أنظمة تشغيل الوقت الحقيقي Real Time Operating Systems والتي تختصر بكلمة RTOS، هذه الأنظمة مشابه في نظرية عملها بنظام التشغيل التقليدي الذي تستخدمة الآن على الحاسب الآلي مثل Windows أو Linux.

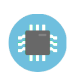
تهدف أنظمة التشغيل إلى توفير مجموعة من الخدمات أهمها "تعدد المهام Multi-tasking" بفضل هذه الأنظمة تستطيع أن تتصفح الويب وفي نفس الوقت سماع ملف صوتي وقد تستخدم برنامج معالجة النصوص بجانب كل ما سبق.

بالنسبة لك أنت تستطيع أن تقوم بكل هذه الأشياء في نفس الوقت لكن الحقيقة أن هذه المهام تتم بالتتباع، أما سرعة تنفيذها فيرجع الفضل في ذلك إلى مهارة نظام التشغيل في معالجة كل هذه المهام بسرعة واستجابة عالية.

كيف تعمل أنظمة التشغيل بشكل عام؟

يعمل نظام التشغيل بصورة مشابهة **للطباخ المحترف**، عادة يطلب من الطباخ أن يعد أكثر من وجبه في نفس الوقت وقد تختلف هذه الوجبات أو تتشابه لكن في النهاية يتوجب عليه إعدادها جميعاً في وقت محدد.





عادة ما يلجأ الطباخ لحيلة ذكية لتنظيم الوقت فهو لا يقوم بإنهاء كل جزء من الوجبه على حدة ثم البدأ في جزء أخر ولكنه بدلاً من ذلك يقوم بعملية التبديل بين المهام Task switching.

فمثلاً إذا كان المطلوب هو إعداد طبق المعكرونة مع اللحم المشوي فأن الطباخ سيبدأ أولاً بغلي الماء ويتركه على النار **(المهمة الأولى)** ثم قد يضع اللحم على الشوايه **(المهمة الثانية)** **ثم** يبدأ في تجهيز الطماطم من أجل الصلصة **(المهمة الثالثة)**، عندما يستشعر أن الماء بدء يغلي بدرجة حرارة مناسبة سيترك الصلصة ثم يعود إلى الماء ليضيف إليه المعكرونة ويتركها حتى تنضج، ثم يعود مرة أخرى للطماطم و في ذلك الوقت فإنه سيحاول كل فترة زمنية أن يطمئن على درجة نضوج اللحم وتستمر هذه العملية حتى تنتهي كل مكونات الوجبة ...

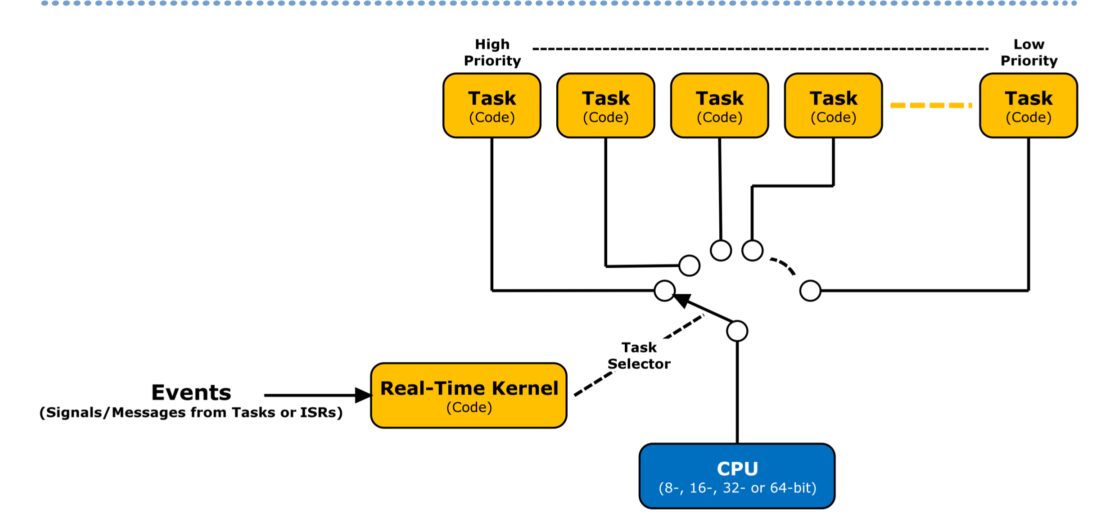
نظام التشغيل يعمل بصورة مشابهه، في البداية يقوم بتجهيز قائمة بالمهمات التي يجب عليه أن ينجزها ثم يبدأ بإدارة المهام مثل الطباخ المحترف، ويقوم بذلك بعدة طرق منها مثلاً أن يعطي كل مهمة فترة معالجة زمنية قصيرة تسمى time slice (شريحة زمنية) وعند انتهاء الشريحة الزمنية يقوم بالتبديل بين المهمات. ويعرف هذا الأسلوب لجدولة المهام بالـ Round Robin scheduling

أيضاً قد يقوم نظام التشغيل بمعالجة المهمات على حسب أولوية إنهائها فمثلاً قد يبدأ في معالجة أهم مهمة متوفرة لفترة طويلة حتى تصله رسالة أو طلب مقاطعة بوجود مهمة أخرى أعلى أهمية و تحتاج لمعالجة فورية فيترك المهمة الأولى ويذهب للثانية مؤقتاً حتى ينهيها ثم يعود للأولى. وعندما ينتهي من المهمة الأولى بالكامل يتركها ويذهب إلى المهمة الثالثه وهكذا ... ويعرف هذا الأسلوب في التبديل بين المهمات بالـ Priotiry based scheduling.

وقبل أن نكمل هناك بعض المسميات الإنجليزية الواجب معرفتها:

يسمى الجزء الذي يدير عمليه التبيدل بين المهمات بالـ RTOS scheduler (مجدول نظام التشغيل) وهو جزء من نواة النظام والتي تسمى RTOS kernel كما تسمى عملية التبديل بين المهمات (بغض النظر عن أسلوب جدولة التبديل) بإسم Context switching

وتسمى المهمة التي يتم معالجتها الآن بالـ Running task بينما المهمة التي تنتظر دورها في المعالجة Waiting task. الصورة التالية طريق عمل الـ RTOS بأسلوب مبسط

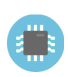


11.3 كيف تعمل النواة RTOS Kernel

المعالجات الموجودة داخل المُتحكِمات الدقيقة تستطيع أن تنفذ مجموعة من الأوامر المتتالية ولا تعرف تحديداً إذا كانت هذه الأوامر جزء من نظام التشغيل أو أنها مهمة معينة وهذا يؤدي إلى سؤال هام، **كيف يفهم المعالج أنه يجب أن يترك المهمة الحالية ويذهب لتنفيذ مهمة أخرى؟**

هنا تأتي وظيفة نواة النظام Kernel . هذه النواة تعمل داخل دالة مقاطعة زمنية Interrupt service routin ويتم تكرار تشغيلها من مئات إلى آلاف المرات في الثانية الواحدة (يسمى عدد مرات تشغيل النواة بال System tick). في كل مرة يتم تشغيل النواة يبدأ جزء من هذه النواة يسمى **"برنامج الجدولة Scheduler"** بمراجعة قائمة المهام الموجودة ويحدد ما الذي سيتم معالجته الآن.

قد يختار برنامج الجدولة أن يكمل المهمة الحالية أو ينتقل إلى مهمة أخرى على حسب خوارزمية التحويل (الـ Round-Robin أو Priority-based) وبعد أن يقرر المهمة يقوم بنقل البيانات الخاصة بمعالجة هذه المهمة إلى المعالج وتتضمن البيانات محتوى المُسجِلات المختلفة (كل من المُسجِلات العامة والخاصة).



11.4 مقدمة عن نظام FreeRTOS

تتوفر العديد من أنظمة تشغيل الوقت الفعلي المجانية والتي تختلف فيما بينها بما تقدمة من إمكانيات ووظائف إضافية، من هذه الأنظمة نجد FreeRTOS والذي يعد أشهر أنظمة الوقت الحقيقي مفتوحة المصدر المتوفرة للإستخدام المجاني سواء بصورة تعليمية أو تجارية.

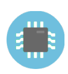


يدعم هذا النظام مختلف المُتحكِمات الدقيقة فيمكنك تشغيله على PIC – ARM – AVR – 8051 – PowerPC – x86 والعديد من المعماريات الأخرى.

كما يوجد منه نسخة مخصصة للأنظمة ذات الموثوقية العالية (مثل الأجهزة الطبية أو التطبيقات العسكرية) ويسمى SAFE-RTOS مع العلم أن هذه النسخة مدفوعة وليست مجانية.

يمتلك هذا النظام العديد من المميزات الرائعة منها:

- حجم صغير يستهلك أقل من 4 إلى 9 كيلو بايت من ذاكرة الـ ROM مما يجعله مناسب لمعظم المُتحكِمات الدقيقة (تذكر أن ATmega16 يمتلك 16 كيلوبايت من الذاكرة بينما يمتلك ATmega32 نحو 32 كيلوبايت).
- توفير مجموعة من المكتبات البرمجية الجاهزة للتعامل مع أنظمة الملفات FAT والـ storage media مثل بطاقات الذاكرة
- مدمج معه مكتبات خاصة لتسهيل معالجة البيانات القادمة من وإلى شبكات الحاسب الآلي عبر بروتوكول TCP/IP أو UDP مما يجعله نظام مناسب لتطبيقات إنترنت الأشياء IoT والتحكم عن بعد



11.5 الهيكل البرمجي لـ RTOS

بصورة افتراضية تكتب جميع أوامر برامج النظم المدمجة في الدالة الرئيسية على عكس أنظمة الـ RTOS حيث نجد أن كل مهمة task تماثل دالة رئيسية مستقله وتكون مهمة الـ Main Function هي تعريف عدد وخصائص المهمات فقط.

الهيكل التالي شكل التقليدي للبرامج المكتوبة باستخدام RTOS حيث تم إنشاء 3 مهام مختلفة وتمتلك كل مهمة مجموعة الأوامر الخاصة بها.

```
#include "RTOS.h"

int main()
{
 createTask( task1, task1_parameters);          // إنشاء المهمة الأولى
 createTask( task2, task2_parameters);          // إنشاء المهمة الثانية
 createTask( task3, task3_parameters);          // إنشاء المهمة الثالثة

 startRTOS_scheduler();                         // إبدء إدارة المهمات بإستخدام RTOS

  while(1) ;                  // لا تفعل أي شيء أخر داخل الدالة الرئيسية إلى ما لا نهاية
  return 0;
}
```

```
void task1( ) {

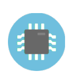
while(1) {
                    // هنا تكتب مجموعة الأوامر الخاصة بالمهمة الأولى
          }
}
```

```
void task2( ) {

 while(1) {
                    // هنا تكتب مجموعة الأوامر الخاصة بالمهمة الثانية
          }
}
```

```
void task3( ) {

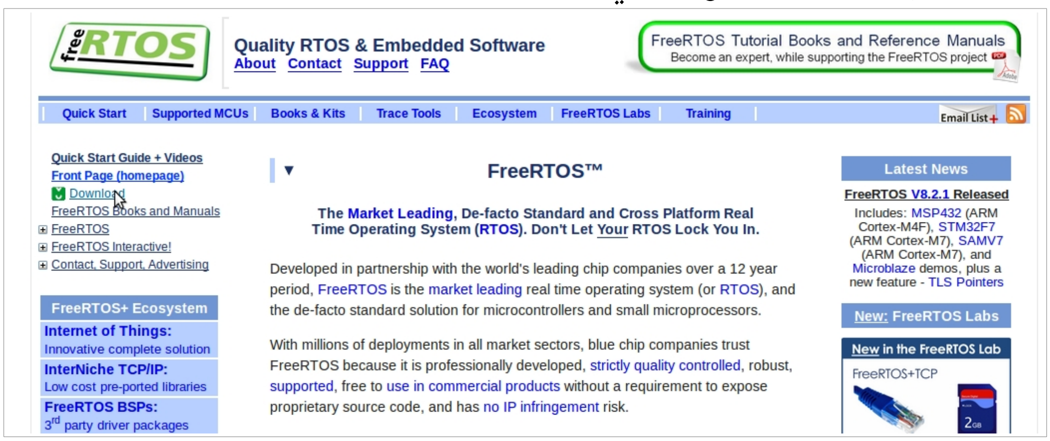
 while(1) {
                    // هنا تكتب مجموعة الأوامر الخاصة بالمهمة الثالثة
          }
}
```



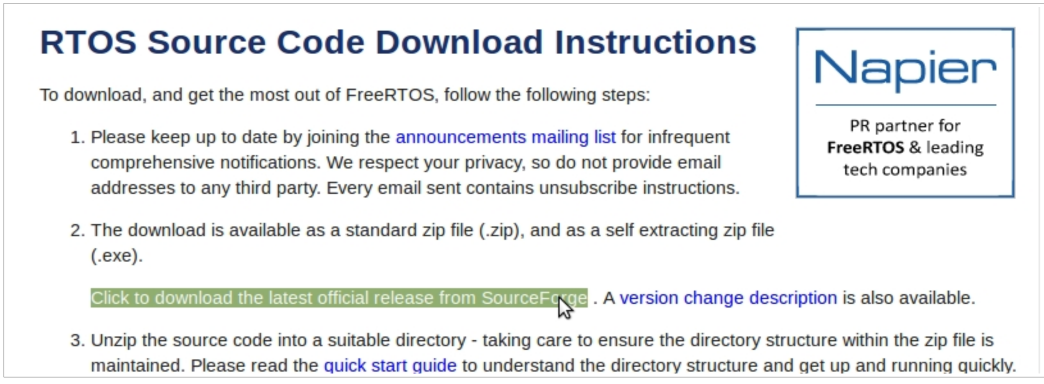
11.6 تشغيل FreeRTOS على جميع مُتحكِمات AVR

عند تحميل نظام FreeRTOS من الموقع الرسمي سنجد أنه مهيأ للعمل على المُتحكِم الدقيق **atmega323** فقط.لذا سنقوم بعمل ما يسمى " **تصدير النظام" system porting** (البعض يسميها تهيئة النظام)، والتي تعني ضبط النظام ليعمل مع مختلف المُتحكِمات الدقيق الأخرى من عائلة AVR. وسنأخذ المُتحكِمات atmega 16/ 32 /328/ 644/1284 مثال على ذلك.

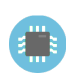
في البداية قم بتحميل أحدث إصدار متوفر من نظام FreeRTOS (وقت كتابة هذا الفصل كان الإصدار رقم 8.2.1) من الموقع الرسمي http://www.freertos.org



اضغط على Download من الجانب الأيسر للموقع للانتقال لصفحة التحميل ثم اختر "تحميل الإصدار الأخير من موقع sourceforge.com" كما في الصورة التالية



عند الانتهاء من التحميل قم بفك ضغط الملف لتجد المجلد الرئيسي باسم FreeRTOS وبه جميع المجلد والملفات الخاصة بنظام التشغيل لمختلف المُتحكِمات الدقيقة



تنقسم الملفات إلى مجلدين أساسين وهما source و Demo. الأول يحتوي على جميع الملفات المصدرية لنظام التشغيل التشغيل نفسه بينما الآخر يحتوي على أمثلة لاختبار نظام التشغيل.

ملاحظة: الملفات تشمل نسخة من النظام + أمثلة لجميع المُتحكِمات من جميع العائلات التي يدعمها نظام FreeRTOS بما في ذلك مُتحكِمات PIC, AVR, ARM cortex ما يهمنا من هذه الملفات ما هو متعلق بال AVR فقط باعتباره موضوع الكتاب.
أيضاً يدعم نظام FreeRTOS العديد من المترجمات المختلفة مثل GCC – IAR – Keil ولكننا سنستخدم GCC فقط باعتباره المترجم المرفق مع ال toolchain المجانية.

تعتمد عملية تصدير النظام على التلاعب بإعدادات FreeRTOS من خلال 3 ملفات رئيسية:

```
port.c
FreeRTOSConfig.h
makefile
main.c
```

تتواجد هذه الملفات في المسارات التالية:

```
FreeRTOS/Source/portable/GCC/ATMega323/port.c
FreeRTOS/Demo/AVR_ATMega323_WinAVR/FreeRTOSConfig.h
FreeRTOS/Demo/AVR_ATMega323_WinAVR/makefile
```

أولاً: تعديل port.c

سمي هذا الملف port.c باعتباره الملف الأساسي في عملية الـ porting حيث يتحكم بالخصائص الأساسية لعملية ال context switching (التبديل بين المهمات tasks). ويتم ضبط إعدادات المؤقت والمقاطعات الدورية من داخل الملف.

يعتمد نظام FreeRTOS على المؤقت رقم 1 في جميع عائلات AVR وبالتحديد يقوم بالتنقل بين المهمات عندما يحدث مقاطعة نتيجة تطابق عداد المؤقت 1 والمسجل OCRA1x وهو ما يعرف باسم timer 1 COMPARE A match interrupt.

في البداية توجه إلى الملف المتواجد داخل المسار التالي

```
FreeRTOS/Source/portable/GCC/ATMega323/port.c
```

افتح الملف بأي محرر نصوص متوفر لديك وتوجه إلى السطر الذي يحتوى العبارة التالية

```
#define portCOMPARE_MATCH_A_INTERRUPT_ENABLE  ( ( uint8_t ) 0x10 )
```

```
#include "task.h"

/*-----------------------------------------------------------
 * Implementation of functions defined in portable.h for the AVR port.
 *----------------------------------------------------------*/

/* Start tasks with interrupts enables. */
#define portFLAGS_INT_ENABLED                    ( ( StackType_t ) 0x80 )

/* Hardware constants for timer 1. */
#define portCLEAR_COUNTER_ON_MATCH               ( ( uint8_t ) 0x08 )
#define portPRESCALE_64                          ( ( uint8_t ) 0x03 )
#define portCLOCK_PRESCALER                      ( ( uint32_t ) 64 )
#define portCOMPARE_MATCH_A_INTERRUPT_ENABLE     ( ( uint8_t ) 0x10 )

/*-----------------------------------------------------------*/
```

أول ما يجب ضبطه هو قيمة الـ mask الذي سيفعل خاصية المقاطعة timer 1 compare match A ويتم حساب هذا الـ mask بالنظر إلى المُسجِل TIMSK في مُتحكِمات AVR القديمة نسبياً أو TIMSK0 في المُتحكِمات الأحدث، يحتوي هذا المُسجِل على البت المطلوب تفعيلها **OCIE1A** والمسؤولة عن تفعيل المقاطعة Timer1 Compare A Match interrupt

القيمة 0x10 تمثل قيمة الـ mask المُتحكِم في تفعيل المقاطعة ويتم حساب هذه القيمة كالتالي:

- توجه إلى ملف البيانات الخاصة بالمتحكم الذي ستضع عليه نظام التشغيل (على سبيل المثال المُتحكِم ATmega32 – صفحة دليل البيانات رقم 82). سنجد أن البت **OCIE1A** هي البت رقم 4

Timer/Counter Interrupt Mask Register – TIMSK

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 1 – OCIE0: Timer/Counter0 Output Compare Match Interrupt Enable**

When the OCIE0 bit is written to one, and the I-bit in the Status Register is set (one), the

- الآن سنقوم بحساب الـ mask وذلك عن طرق وضع الرقم 1 مكان هذه البت وباقي البتات =0 وهو ما يعني الرقم 00010000 بالصيغة الثنائية وهو ما يساوي **0x10** بصيغة الهيكس. (هذه القيمة هي الافتراضية بالفعل)

في حالة المُتحكِم **atmega168** أو **atmega328** **(ومعظم المُتحكِمات الحديثة)** سنجد أن مكان البت **OCIE1A** مختلف. فمثلاً في حالة المُتحكِم atmega328 سنجد الوضع التالي (صفحة 135 من دليل البيانات للمُتحكِم atmega328).

TIMSK1 – Timer/Counter1 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

في حالة المُتحكِم **atmega644** أو المُتحكِم **atmega1284** صفحة 134 من دليل البيانات

TIMSK1 – Timer/Counter1 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

إذا قمنا بوضع الرقم 1 مكان هذه البت وباقي البتات = صفر سنجد أن قيمة الـ mask تساوي 00000010 وهو ما يساوي 0x02 بصيغة الهيكس. إذا يجب استبدال القيمة 0x10 بالقيمة 0x02 كالتالي:

```
#define portPRESCALE_64                       ( ( uint8_t ) 0x03 )
#define portCLOCK_PRESCALER                   ( ( uint32_t ) 64 )
#define portCOMPARE_MATCH_A_INTERRUPT_ENABLE  ( ( uint8_t ) 0x02 )
```

لاحظ أن المُتحكِمات الحديثة نسبياً من عائلة AVR – 8bit لا تمتلك المُسجِل TIMSK وإنما يكون باسم TIMSK0 أو TIMSK1 كما في حالة atmega328 و atmega1284

بعد تعديل قيمة الـ mask سنقوم بتعديل بعض أسماء المُسجِلات في نفس الملف وبالتحديد في الدالة **prvSetupTimerInterrupt(void)** والتي ستجدها في أخر الملف port.c كما في الصورة التالية:

```
static void prvSetupTimerInterrupt( void )
{
uint32_t ulCompareMatch;
uint8_t ucHighByte, ucLowByte;

    /* Using 16bit timer 1 to generate the tick.  Correct fuses must be
    selected for the configCPU_CLOCK_HZ clock. */

    ulCompareMatch = configCPU_CLOCK_HZ / configTICK_RATE_HZ;

    /* We only have 16 bits so have to scale to get our required tick rate. */
    ulCompareMatch /= portCLOCK_PRESCALER;

    /* Adjust for correct value. */
    ulCompareMatch -= ( uint32_t ) 1;

    /* Setup compare match value for compare match A.  Interrupts are disabled
    before this is called so we need not worry here. */
    ucLowByte = ( uint8_t ) ( ulCompareMatch & ( uint32_t ) 0xff );
    ulCompareMatch >>= 8;
    ucHighByte = ( uint8_t ) ( ulCompareMatch & ( uint32_t ) 0xff );
    OCR1AH = ucHighByte;
    OCR1AL = ucLowByte;

    /* Setup clock source and compare match behaviour. */
    ucLowByte = portCLEAR_COUNTER_ON_MATCH | portPRESCALE_64;
    TCCR1B = ucLowByte;

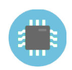
    /* Enable the interrupt - this is okay as interrupt are currently globally
    disabled. */
    ucLowByte = TIMSK;
    ucLowByte |= portCOMPARE_MATCH_A_INTERRUPT_ENABLE;
    TIMSK = ucLowByte;
}
```

إذا كان المُسجِل المسؤول عن المقاطعة يسمى TIMSK فلن تغير أي شيء بها (مثل المُتحكِم ATmega32/16) أما إذا كنت ستستخدم أي مُتحكِم حديث مثل atmega328 أو atmega1284 يجب أن تستبدل TIMSK بالاسم المناسب مثل TIMSK0 أو TIMSK1 (على حسب ما هو مذكور في دليل البيانات كما شاهدنا في الخطوة السابقة).

مثال: تعديل الدالة لتتناسب مع المُتحكِم atmega1284/644/328/168/....etc

```
    /* Enable the interrupt - this is okay as interrupt are currently globally
    disabled. */
    ucLowByte = TIMSK1;
    ucLowByte |= portCOMPARE_MATCH_A_INTERRUPT_ENABLE;
    TIMSK1 = ucLowByte;
}
/*-----------------------------------------------------------*/
```

الجزء الأخير من تعديل ملف port.c هو إعادة تسمية الـ ISR الخاصة بمقاطعة Timer1 Compare A Match وهذا الجزء يجب تغييره لمعظم مُتحكِمات AVR (سواء القديمة أو الحديثة). هذا الجزء ستجده في نهاية ملف port.c كما في الصورة التالية.



```
#if configUSE_PREEMPTION == 1

    /*
     * Tick ISR for preemptive scheduler.  We can use a naked attribute as
     * the context is saved at the start of vPortYieldFromTick().  The tick
     * count is incremented after the context is saved.
     */
    void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal, naked ) );
    void SIG_OUTPUT_COMPARE1A( void )
    {
        vPortYieldFromTick();
        asm volatile ( "reti" );
    }
#else

    /*
     * Tick ISR for the cooperative scheduler.  All this does is increment the
     * tick count.  We don't need to switch context, this can only be done by
     * manual calls to taskYIELD();
     */
    void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal ) );
    void SIG_OUTPUT_COMPARE1A( void )
    {
        xTaskIncrementTick();
    }
#endif
```

قم بتغير كل الدوال باسم **SIG_OUTPUT_COMPARE1A**

إلى الاسم **TIMER1_COMPA_vect**

ليصبح الملف كالتالي:

```
    void TIMER1_COMPA_vect( void ) __attribute__ ( ( signal, naked ) );
    void TIMER1_COMPA_vect( void )
    {
        vPortYieldFromTick();
        asm volatile ( "reti" );
    }
#else

    /*
     * Tick ISR for the cooperative scheduler.  All this does is increment the
     * tick count.  We don't need to switch context, this can only be done by
     * manual calls to taskYIELD();
     */
    void TIMER1_COMPA_vect( void ) __attribute__ ( ( signal ) );
    void TIMER1_COMPA_vect( void )
    {
        xTaskIncrementTick();
    }
#endif
```

بذلك نكون قد انتهينا من تعديل الملف port.c

ثانيا: تعديل FreeRTOSConfig.h

يتحكم هذا الملف في إعدادات نظام التشغيل عندما يبدأ العمل على المُتحكِم الدقيق، وبالتحديد هو المسؤول عن التلاعب بخصائص الـ kernel وبعض خصائص تنظيم الـ tasks. يمكنك استخدام الملف كما هو بالوضع الافتراضي لكن ستحتاج أن تغير بعض الإعدادات الهامة لتناسب المُتحكِم الدقيق المستخدم. وبعض الإعدادات الأخرى والتي ستجعل نظام التشغيل أكثر كفاءة في إدارة المهام (threads) tasks خاصة مع المُتحكِمات ذات الذاكرة الصغيرة.

يتواجد الملف في نفس مجلد Demo بجانب الملف main.c وبالتحديد في المسار التالي

```
FreeRTOS/Demo/AVR_ATMega323_WinAVR/FreeRTOSConfig.h
```

الإعدادات الواجب ضبطها

```
#define configCPU_CLOCK_HZ                  ( ( unsigned long ) 8000000 )
```

يتحكم هذا الخيار في ضبط التوقيت داخل نظام التشغيل، يجب أن تغير قيمته إلى نفس التردد الذي يعمل به المعالج (القيمة الافتراضية هي 8 ميجاهرتز).

```
#define configTICK_RATE_HZ                  ( ( TickType_t ) 1000 )
```

عدد المرات التي تعمل بها الـ kernel وذلك التبديل بين المهمات المختلفة. فمثلاً بافتراض أن لديك 10 مهمات فهذا يعني أن الـ Kernel ستقوم بالتبديل بينهم 1000 مرة وهو ما يعني أن كل مهمة سيتم زيارتها 100 مرة في الثانية في الواحد.

زيادة هذا الرقم ستؤدي إلى نقص الوقت المخصص للمعالجة لكل مهمة في مقابل تحسين الاستجابة للمهمات ككل، ولكن هناك عيب واحد وهو أن الـ kernel سيزداد معدل استهلاكها للوقت والموارد بزيادة هذا الرقم. فمثلا إذا كان معدل الـ tick = 3000 Hz فهذا يعني أن الـ kernel ستعمل 3000 مرة في الثانية وفي كل مرة قد تختار أحد المهمات لتشغيلها أو التبديل مع مهمة أخرى context switch.

من المفيد زيادة هذا الرقم في حالة أن المهمات تحتاج لاستجابة سريعة لكن يستحسن أن يكون المُتحكِم يعمل بسرعة 16 أو 20 ميجاهرتز على الأقل. مع العلم أنه بزيادة هذا الرقم

سيتم استهلاك المزيد من الطاقة. لذا إن كان مشروعك يجب تصميمه باستهلاك طاقة منخفض فيستحسن تقليل الرقم إلى 500 (سيخفض سرعة الاستجابة للمهمات) وإذا كنت لا تهتم بزيادة استهلاك الطاقة فيمكنك أن تضع القيمة بين 1000 و 3000 وفي حالة المُتحكِمات من عائلة AVR Xmega أو AVR – 32 bit يمكن زيادة الرقم إلى 5000.

```
#define configMAX_PRIORITIES                ( 4 )
```

هذا الخيار يتحكم بأقصى عدد من الأولويات المسموح بها في نظام التشغيل. يتم تحديده بناء على عدد المهمات المتوقع تشغيلها على النظام ومدى أهمية كل منها. إذا كانت معظم المهمات لها نفس الأولوية فيمكنك أن تضع الرقم ب **2** فقط.

```
#define configMINIMAL_STACK_SIZE            ( ( unsigned short ) 85 )
```

الرقم المسؤول عن تحديد أقل حجم لـ stack لكل مهمة تعمل، يجب أن يتناسب الرقم مع حجم الذاكرة العشوائية SRAM للمُتحكِم الدقيق. فمثلاً إذا كانت ال SRAM تساوى 1 كيلو فمن الأفضل أن يصبح الرقم 50 فقط أما إذا كانت الذاكرة 2 كيلو فيمكنك أن تتركه كما هو أما إذا كانت الذاكرة 4 كيلو أو أكثر فمن الممكن زيادتها إلى 150.

أيضاً يجب زيادة هذا الرقم إذا كانت المهمات التي سيتم تشغيلها ستتعامل مع دوال أو عمليات حسابية كبيرة (أو أي عملية تحتاج أن تستخدم الـ Stack لإجرائها). ولاحظ أن معظم العمليات الحسابية المركبة مثل x = y+5/z*(x+1) تستخدم الـ stack لذا يجب أن تضع هذه العوامل بالحسبان عند اختيار هذا الرقم. أيضاً يتم استخدام الـ stack في حفظ قيم المُسجِلات التي تتعامل معها المهمة.

يجب الانتباه إلى أمر هام وهو أن حجم ال stack لكل مهمة يقاس بال word وليس بال byte لذا عندما نقول أن حجم ال stack = 150 فهذا يعني أن الحجم الحقيقي في الذاكرة

stack size= 150 word = 150 * (4 byte “1 Word”) = 600 byte

```
#define configTOTAL_HEAP_SIZE               ( (size_t ) ( 1500 ) )
```

الـ Heap هي تقنية data-type تستخدم أنظمة RTOS لتخصيص ذاكرة مرنة (قابلة للزيادة) لكل مهمة تعمل في نظام التشغيل. يتم استخدام هذه التقنية في حفظ جميع المتغيرات أو الثوابت المستخدمة في كل مهمة.

يتم تحديد هذا الرقم بصورة أساسية على حسب حجم الذاكرة العشوائية الكلية للمُتحكِم الدقيق ويجب الانتباه أن الرقم 1500 يساوي قيمة الذاكرة بالبايت byte وهذا يعني أن ذاكرة الـ heap هنا تساوي الخامس كيلو بايت تقريباً.

يمكن تحديد هذا الرقم بما هو إجمالي الذاكرة العشوائية للمُتحكِم الدقيق مع مراعاة أن نظام freeRTOS يحتاج نحو 300 إلى 400 بايت لتشغيل الـ kernel بينما يمكن تخصيص باقي الذاكرة للمهمات المختلفة.

الأمثلة التالية هي لقيم الـ heap للمتحكمات المختلفة

- المُتحكِم **ATmega16** يمتلك SRAM = 1 Kbyte
- مساحة الـ heap يفضل أن تكون **0.8 كيلوبايت (800).**

- المُتحكِمات **ATmega32/328** تمتلك SRAM = 2 Kbyte
- مساحة الـ heap يفضل أن تكون **واحد ونصف كيلوبايت (1500)**

- المُتحكِم **atmega644** يمتلك SRAM = 4 Kbyte
- مساحة الـ heap يفضل أن تكون **3.5 كيلوبايت (3584)**

- المُتحكِم **atmega1284** يمتلك SRAM = 16 Kbyte
- مساحة الـ heap يفضل أن تكون **15.5 كيلوبايت (15360)**

بذلك نكون قد انتهينا من أهم الإعدادات للنظام
يمكنك أن تتعلم حول باقي الإعدادات من المرجع المفصل لملف FreeRTOSCOnfig.h على الرابط التالي http://www.freertos.org/a0019.html

ثالثاً: تعديل serial.c (اختياري - غير مطلوب)

يوفر نظام FreeRTOS ملف serial.c وهو driver لتشغيل الـ UART داخل نظام التشغيل ويعتبر متوافق مع معظم مُتحكِمات AVR 8 bit لكن يحتاج بعض التعديلات البسيطة وهي

تغيير أسماء المُسجِلات الداخلية المستخدمة في الملف (أيضاً بسبب اختلاف المُتحكِمات الحديثة عن القديمة في تسمية المُسجِلات مثل UCSRB أصبح UCSR0B أو UCSR1B).
التعديلات المقترحة (على حسب المُتحكِم المستخدم) - يجب مراجعة دليل البيانات الجزء الخاص بال USART لمعرفة أسماء المُسجِلات

مثال: تعديل الـ driver ليتوافق مع المُتحكِم atmega328 (يجب استبدال جميع المُسجِلات بالأسماء الصحيحة لها).

UCSRB -> **UCSR0B**
UCSRC -> **UCSR0C**
UBRRL -> **UBRR0L**
UBRRH -> **UBRR0H**
UDR -> **UDR0**

ملاحظة: لن يتم استخدام هذا الملف في جميع الأمثلة الموجودة في الكتاب والتعديلات المذكورة بالأعلى اختيارية في حالة أنك تريد استخدام الـ UART في مشاريعك الخاصة

رابعاً: تعديل makefile

التعديلات في هذا الملف مرتبطة بمستخدمي نظام Linux أو أي شخص يُستخدَم الـ toolchain مباشرة دون استخدام بيئة برمجة متكاملة مثل (ATmel studio) IDE، يمكنك قراءة الملحق الخاص بعملية compiling using makefile لتتعرف أكثر على هذه التقنية.

يوجد الملف makefile في نفس مجلد Demo ويحتوي على بيانات المُتحكِم والمبرمجة التي سيتم استخدامها، حيث يجب ضبط هذه الإعدادات لتتوافق مع المُتحكِم المستخدم.

تغيير المُتحكِم

في بداية الملف ستجد السطر البرمجي MCU = atmega323 قم بتغييره لاسم المُتحكِم المطلوب مثل ATmega16 أو ATmega32 أو atmega1284 كما في الصورة التالية

```
# MCU name
MCU = atmega32

```

تعديل المكتبات المُضافة

في الجزء الخاصة بالـ compiling سنجد أن ملف makefile الافتراضي يتضمن العديد من المجلدات التي تحتوي جميع المكتبات المطلوبة لتشغيل نظام FreeRTOS، يمكنك إضافة مكتباتك الخاصة أو مسح المكتبات التي لا تحتاجها من هذا الجزء.

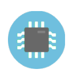
```
# List C source files here. (C dependencies are automatically generated.)
DEMO_DIR = ../Common/Minimal
SOURCE_DIR = ../../Source
PORT_DIR = ../../Source/portable/GCC/ATMega323

SRC = \
main.c \
ParTest/ParTest.c \
serial/serial.c \
regtest.c \
$(SOURCE_DIR)/tasks.c \
$(SOURCE_DIR)/queue.c \
$(SOURCE_DIR)/list.c \
$(SOURCE_DIR)/croutine.c \
$(SOURCE_DIR)/portable/MemMang/heap_1.c \
$(PORT_DIR)/port.c \
$(DEMO_DIR)/crflash.c \
$(DEMO_DIR)/integer.c \
$(DEMO_DIR)/PollQ.c \
$(DEMO_DIR)/comtest.c
```

في الأمثلة المذكورة في الكتاب لا داعي لإضافة كل هذه المكتبات لذا سنقوم بمسحها واستبدالها بالقائمة التالية فقط

```
SRC     = \
main.c \
$(SOURCE_DIR)/tasks.c \
$(SOURCE_DIR)/queue.c \
$(SOURCE_DIR)/list.c \
$(SOURCE_DIR)/croutine.c \
$(SOURCE_DIR)/timers.c \
$(SOURCE_DIR)/portable/MemMang/heap_1.c \
$(PORT_DIR)/port.c
```

المكتبات المُضافة بعد التعديل



```
# List C source files here. (C dependencies are automatically generated.)
DEMO_DIR = ../Common/Minimal
SOURCE_DIR = ../../Source
PORT_DIR = ../../Source/portable/GCC/ATMega323

SRC = \
main.c \
$(SOURCE_DIR)/tasks.c \
$(SOURCE_DIR)/queue.c \
$(SOURCE_DIR)/list.c \
$(SOURCE_DIR)/croutine.c \
$(SOURCE_DIR)/timers.c \
$(SOURCE_DIR)/portable/MemMang/heap_1.c \
$(PORT_DIR)/port.c
```

تعديل المُبرمجة

هنا سنقوم باختيار المُبرمجة المستخدمة في حرق ملف الهيكس، ابحث عن العبارة التالية

AVRDUDE_PROGRAMMER = stk500

شخصياً أستخدم usbasp لذا سأقوم باستبدال stk500 بكلمة usbasp (استبدلها باسم المُبرمجة التي لديك إذا كانت مختلفة عن usbasp) كما في الصورة التالية

```
AVRDUDE_PROGRAMMER = usbasp


AVRDUDE_PORT = com1    # programmer connected to serial device
#AVRDUDE_PORT = lpt1    # programmer connected to parallel port

AVRDUDE_WRITE_FLASH = -U flash:w:$(TARGET).hex
#AVRDUDE_WRITE_EEPROM = -U eeprom:w:$(TARGET).eep

AVRDUDE_FLAGS = -p $(MCU) -P $(AVRDUDE_PORT) -c $(AVRDUDE_PROGRAMMER)
```

ملاحظة: في حالة أنك ستستخدم usbasp يجب أن تمسح الخيار

-P $(AVRDUDE_PORT)

أما في حالة استخدام أي مبرمجة أخرى تمتلك بورت على نظام ويندوز أو لينكس مثل ISP usbTiny أو mkII AVRISP فيجب أن تحدد رقم البورت في الخيار المسمى AVRDUDE_PORT

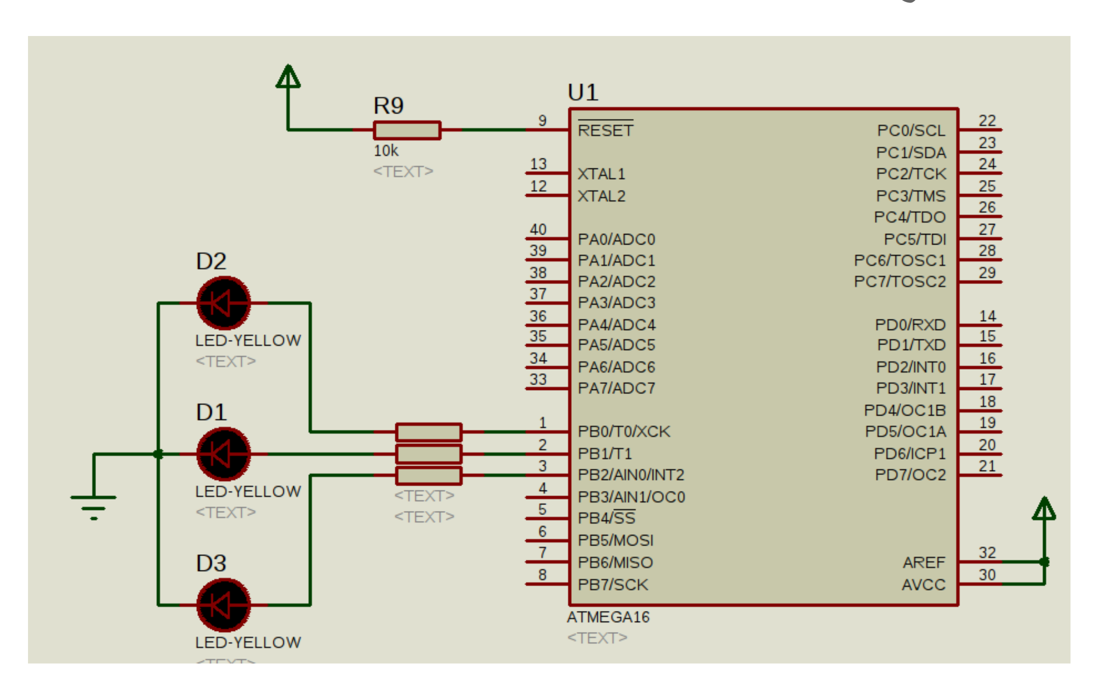
```

AVRDUDE_FLAGS = -p $(MCU) -P $(AVRDUDE_PORT) -c $(AVRDUDE_PROGRAMMER)
```

11.7 المثال الأول: Blinking 3 leds with 3 tasks

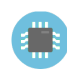
يأتي نظام FreeRTOS بمثال برمجي جاهز لاختبار تعدد المهام لكنه معقد قليلاً لذا أفضل أن نبدأ مع مثال أبسط و أكثر وضوحاً وهو عبارة عن تشغيل 3 ليدات مختلفة كل منها تعمل بتوقيت مختلف عن الآخرى. الليدات متصلة بالترتيب على PB0, PB1, PB2 كما هو موضح في الصورة التالية (المثال تم باستخدام ATmega16 ويمكن استخدام ATmega32 بنفس التوصيلات أيضاً).

مخطط الدائرة



قم بفتح الملف main.c في مجلد Demo وامسح الأكواد البرمجية الموجودة به واستبدلها بالأكواد التالية

> ملاحظة: الأكواد مرفقة في الملف FreeRTOS Examples/blinking_3_leds.c
> أيضاً يمكنك استبدال الملف مباشرة بالملف blinking_3_leds.c مع تغير اسمه إلى main.c ليصبح هو الملف الرئيسي الذي سيتم ترجمته



الكود

```
#define F_CPU 8000000UL
#include <avr/io.h>
#include <util/delay.h>

/* FreeRTOS files. */
#include "FreeRTOS.h"
#include "task.h"
#include "croutine.h"

/* Define all the tasks */
static void ledBlinkingtask1(void* pvParameters);
static void ledBlinkingtask2(void* pvParameters);
static void ledBlinkingtask3(void* pvParameters);

int main(void) {

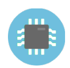
 /* Call FreeRTOS APIs to create tasks, all tasks has the same priority "1" with the
same stack size*/

   xTaskCreate( ledBlinkingtask1, ( signed char * ) "LED1",
configMINIMAL_STACK_SIZE, NULL, 1, NULL );
   xTaskCreate( ledBlinkingtask2, ( signed char * ) "LED2",
configMINIMAL_STACK_SIZE, NULL, 1, NULL );
   xTaskCreate( ledBlinkingtask3, ( signed char * ) "LED3",
configMINIMAL_STACK_SIZE, NULL, 1, NULL );

   // Start the RTOS kernel
         vTaskStartScheduler();

   /* Do nothing here and just run infinte loop */

   while(1){};
   return 0;
}
```



```
static void ledBlinkingtask1(void* pvParameters){

   /* Define all variables related to ledBlinkingtask1*/
   const uint8_t blinkDelay = 50 ;

   /* make PB0 work as output*/
   DDRB |= (1<<PB0);

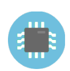
    /* Start the infinte task 1 loop */
    while (1)
    {
      PORTB ^= (1<<PB0);     //toggle PB0
      vTaskDelay(blinkDelay); //wait some time
    }
}
```

```
static void ledBlinkingtask2(void* pvParameters){

   /* Define all variables related to ledBlinkingtask2*/
   const uint8_t blinkDelay = 150;

   /* make PB1 work as output*/
   DDRB |= (1<<PB1);

    /* Start the infinte task 2 loop */
    while (1)
    {
      PORTB ^= (1<<PB1);     //toggle PB0
      vTaskDelay(blinkDelay); //wait some time
    }
}
```



```
static void ledBlinkingtask3(void* pvParameters){

  /* Define all variables related to ledBlinkingtask3*/
  const uint16_t blinkDelay = 600;

  /* make PB2 work as output*/
  DDRB |= (1<<PB2);

   /* Start the infinte task 3 loop */
   while (1)
   {
      PORTB ^= (1<<PB2);     //toggle PB0
      vTaskDelay(blinkDelay); //wait some time
   }
}
```

صورة للملف main.c بعد تعديل الأكواد

```
#define F_CPU 8000000UL
#include <avr/io.h>
#include <util/delay.h>

/* FreeRTOS files. */
#include "FreeRTOS.h"
#include "task.h"
#include "croutine.h"

/* Define all the tasks */
static void ledBlinkingtask1(void* pvParameters);
static void ledBlinkingtask2(void* pvParameters);
static void ledBlinkingtask3(void* pvParameters);

int main(void){

    /* Call FreeRTOS APIs to creat tasks, all tasks has the same priority "1" with the same stack size*/
    xTaskCreate( ledBlinkingtask1, ( signed char * ) "LED1", configMINIMAL_STACK_SIZE, NULL, 1, NULL );
    xTaskCreate( ledBlinkingtask2, ( signed char * ) "LED2", configMINIMAL_STACK_SIZE, NULL, 1, NULL );
    xTaskCreate( ledBlinkingtask3, ( signed char * ) "LED3", configMINIMAL_STACK_SIZE, NULL, 1, NULL );

    // Start the RTOS kernel
    vTaskStartScheduler();
    /* Do nothing here and just run infinte loop */

    while(1){};
    return 0;
}
```

بعد الانتهاء من جميع التعديلات احفظ الملف ثم قم بفتح سطر الأوامر في نفس المجلد واكتب الأمر make لتبدأ عملية الترجمة، إذا تمت بنجاح ستظهر نتائج مشابهه للصورة التالية:

```
avr-objcopy: --change-section-lma .eeprom=0x0000000000000000 never used

Creating Extended Listing: rtosdemo.lss
avr-objdump -h -S rtosdemo.elf > rtosdemo.lss

Creating Symbol Table: rtosdemo.sym
avr-nm -n rtosdemo.elf > rtosdemo.sym

Size after:
rtosdemo.elf  :
section      size      addr
.data          22   8388864
.text        8000         0
.bss         1651   8388886
.stab       27744         0
.stabstr    13082         0
.comment       17         0
Total       50516



Errors: none
-------- end --------
```

الآن يمكنك استخدام ملف الهيكس الناتج من عملية الترجمة سواء في تجربة الـ simulation على برنامج بروتس أو تجربة الملف مباشرة على المُتحكِم الدقيق ATmega16.

في حالة استخدام مُتحكِم أخر مثل ATmega32 أو atmega328 لا تنسى أن تغير الإعدادات الخاصة بالملف makefile – كما هو مذكور في الخطوات في بداية الفصل

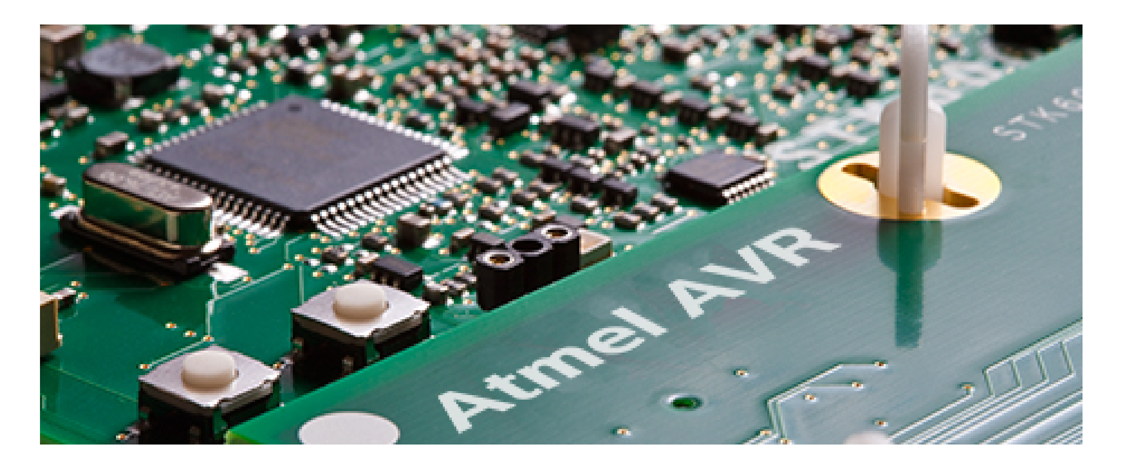
ملاحظة: بالرغم من إمكانية تصميم الأنظمة الـ Hard Real time أو Firm Real time بإستخدام المُتحكِمات الدقيقة مع RTOS إلا أنه عادة ما يتم استخدام تقنية الـ FPGA أو الـ ASIC لتصميم هذه الأنظمة حيث تتمتع هذه التقنيات بالقدرة على معالجة العديد من المهمات المختلفة والاستجابة الفائقة لها جميعاً في نفس الوقت. بينما المُتحكِمات الدقيقة لا تستطيع أن تشغل أكثر من مهمة في نفس اللحظة حتى وإن كان باستطاعتها التبديل بين المهمات بإستخدام الـ RTOS وهذا لأن أغلب المُتحكِمات الدقيقة تمتلك نواة معالجة واحدة Single CPU core

الملحقات

" االتعليم أقوى سلاح يمكنك استخدامه لتغيير العالم "

نيلسون مانديلا

12. المُلحقات الإضافية



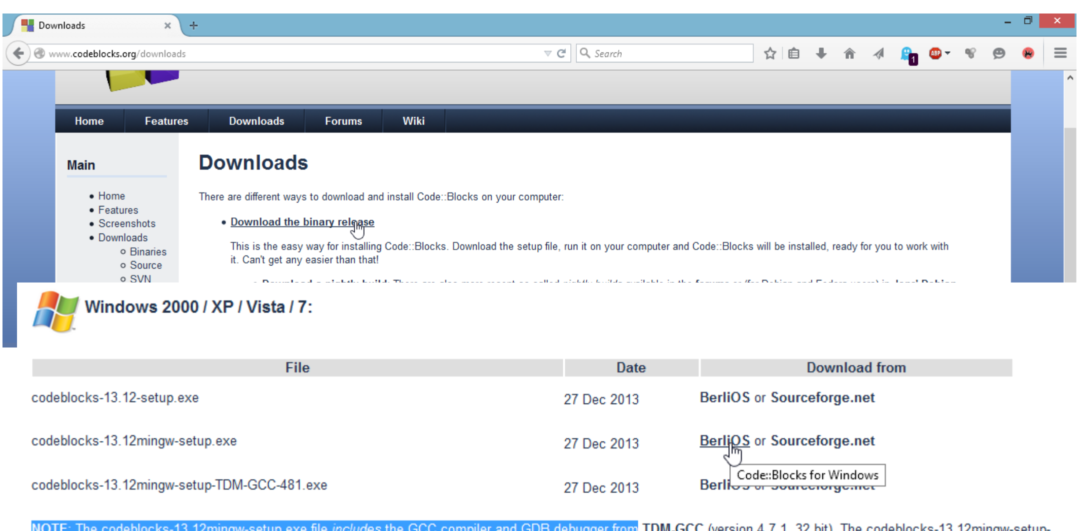
- ✔ تنصيب واستخدام برنامج CodeBlocks كبديل للـ ATmel Studio
- ✔ ترجمة الملفات باستخدام MakeFile
- ✔ رفع ملف الـ Hex على المتحكم الدقيق
- ✔ كيف تستخدم لوحة آردوينو لتعلم برمجة الـ AVR

مُلحق: تنصيب برنامج CodeBlocks على نظام ويندوز

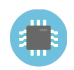
يعتبر برنامج CodeBlocks من أفضل بيئات البرمجة المخصصة للمشاريع التي تكتب بلغة السي أو السي ++ كما أنه مجاني ومتوفر لجميع أنظمة التشغيل (ويندوز - لينكس - ماك)،ويعتبر بديل أخف وأسرع بكثير من ATmel Studio ومناسب لكل من يريد أن يكتب برامج بلغة السي عامة سواء للمتحكمات الدقيقة أو للحواسيب.

يعتمد البرنامج بصورة أساسية على المترجم الشهير GCC والذي سنستخدم الإصدارة المشتقة منه وهي avr-gcc كما سنرى في الخطوات القادمة. في البداية قم بتحميل ملفات التنصيب الخاصة بالبرنامج عبر التوجه إلى موقع CodeBlocks الرسمي من خلال الرابط التالي:

http://www.codeblocks.org/downloads



اضغط على زر تحميل ملفات التشغيل لتظهر لك صفحة تحتوي 3 خيارات للتحميل، قم بتنزيل الخيار الذي يحمل اسم codeblocks-13.12mingw-setup والذي يعني أن برنامج CodeBlocks مضاف إليه جميع ملفات المترجم gcc-compiler (قد يختلف الرقم 13-12 الموجود في اسم البرنامج إذا تم إصدار نسخة جديدة).



بعد الانتهاء من التحميل ابدأ بتنصيب الملف وتأكد من وضع علامة "صح" على جميع خيارات التنصيب الخاصة بالبرنامج (هذه الخيارات تعني تنصيب جميع ملحقات برنامج CodeBlocks كما في الصور التالية:

Code::Blocks Installation

Choose Components

Choose which features of CodeBlocks you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select the type of install: Full: All plugins, all tools, just everything

Or, select the optional components you wish to install:

- Default install
- Contrib Plugins
- C::B Share Config
- MinGW Compiler Suite

Space required: 249.4MB

Description

Position your mouse over a component to see its description.

Nullsoft Install System v2.46

< Back | Next > | Cancel

Code::Blocks Installation

Installing

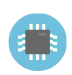
Please wait while CodeBlocks is being installed.

Extract: compiler_dmc.xml... 100%

Extract: codecompletion-off.png... 100%
Output folder: C:\Program Files (x86)\CodeBlocks\share\CodeBlocks
Extract: compiler.zip... 100%
Output folder: C:\Program Files (x86)\CodeBlocks\share\CodeBlocks\plugins
Extract: compiler.dll... 100%
Output folder: C:\Program Files (x86)\CodeBlocks\share\CodeBlocks\compilers
Extract: compiler_avr-gcc.xml... 100%
Extract: compiler_bcc.xml... 100%
Extract: compiler_clang.xml... 100%

Nullsoft Install System v2.46

< Back | Next > | Cancel



تصيب برنامج WinAVR

بعد الانتهاء من تنصيب برنامج CodeBlocks سنقوم بتنزيل الـ WinAVR Toolchain التي تحتوي على المترجم مفتوح المصدر AVR-GCC بالإضافة إلى جميع المكتبات البرمجية + برنامج avrdude

http://sourceforge.net/projects/winavr/files

اضغط على رابط التحميل الموجود بأعلى الصفحة كما في الصورة التالية:

sourceforge | Search | Browse | Enterprise | Blog | Help | Job

SOLUTION CENTERS | Go Parallel | Resources | Newsletters

Home / Browse / Development / Build Tools / WinAVR / Files

WinAVR

Brought to you by: arcanum, joerg_wunsch, sprintersb

Summary | Files | Reviews | Support | Wiki | News | Code | Mailing Lists

Looking for the latest version? Download WinAVR-20100110-install.exe (28.8 MB)

/WinAVR/20100110/WinAVR-20100110-install.exe: released

Home

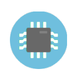
Name	Modified	Size	Downloads / Week
WinAVR	2010-01-20		3,590
Release Candidate	2009-03-07		4

Totals: 2 Items

بعد الانتهاء من التحميل قم بتنصيب البرنامج كما في الخطوات السابقة. والآن سنقوم بتنزل برنامج الواجهة الرسومية AVRDUDESS (إذا لم تكن قمت بتحميلها مسبقاً) تذكر أن برنامج يعمل avrdude لا يعمل إلا من خلال سطر الأوامر فقط لذا سنحتاج AVRDUDESS .

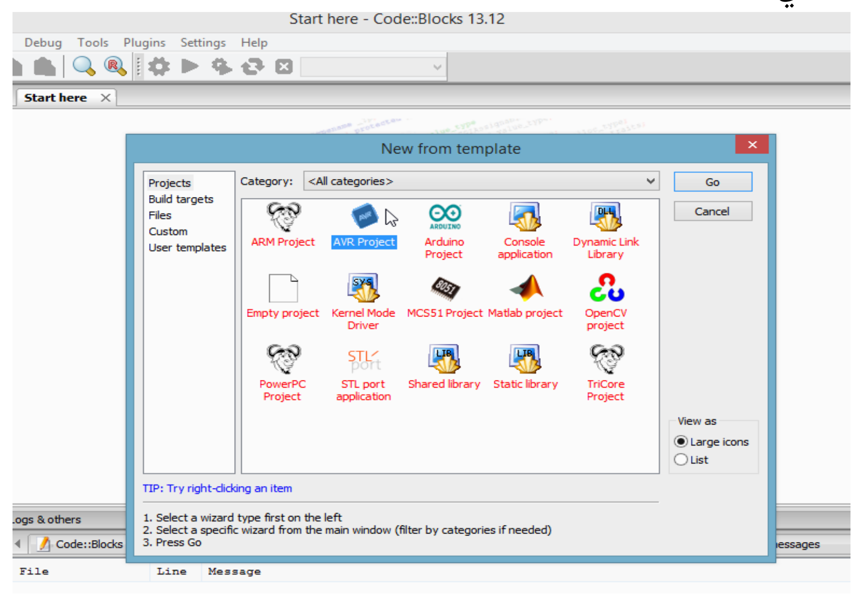
http://blog.zakkemble.co.uk/avrdudess-a-gui-for-avrdude/

ملاحظة: تتوفر العديد من البرامج الأخرى التي تعمل كواجهة رسومية غير AVRdudess مثل: avrdude-gui أو BitBurnner أو AVR8 Burn-O-Mat أيضاً هناك العديد من البرامج الأخرى غير avrdude متوفرة لنظام ويندوز لكني فضلت avrdude لأنه يأتي مع الـ toolchain ويعمل على جميع أنظمة التشغيل

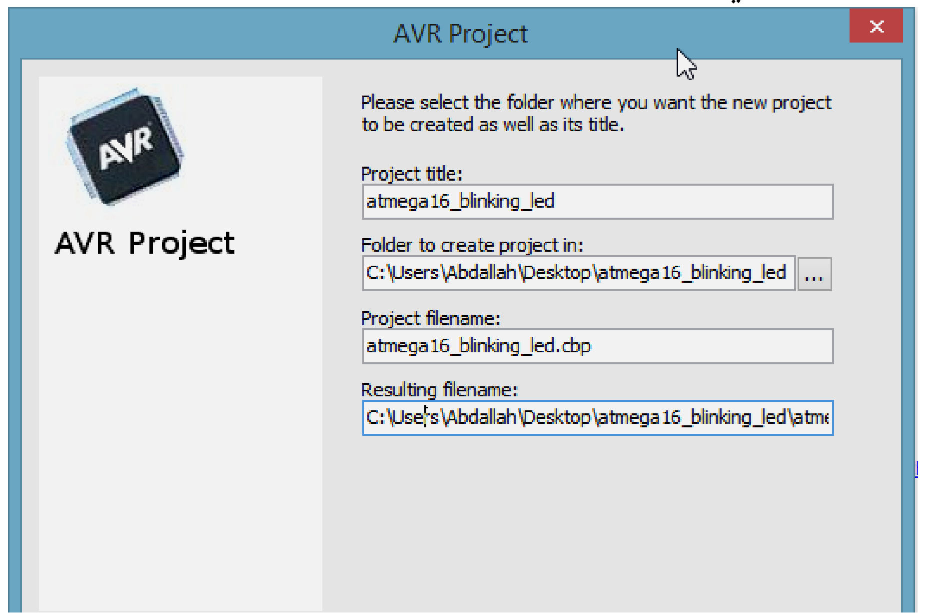


الآن يمكنك البدء في كتابة البرامج لمتحكمات الـ AVR ، لنأخذ الـ Blinking Led كمثال

قم بتشغيل برنامج CodeBlocks واختر New Project ومن الصفحة التي ستظهر اختر AVR project كما في الصور التالية:

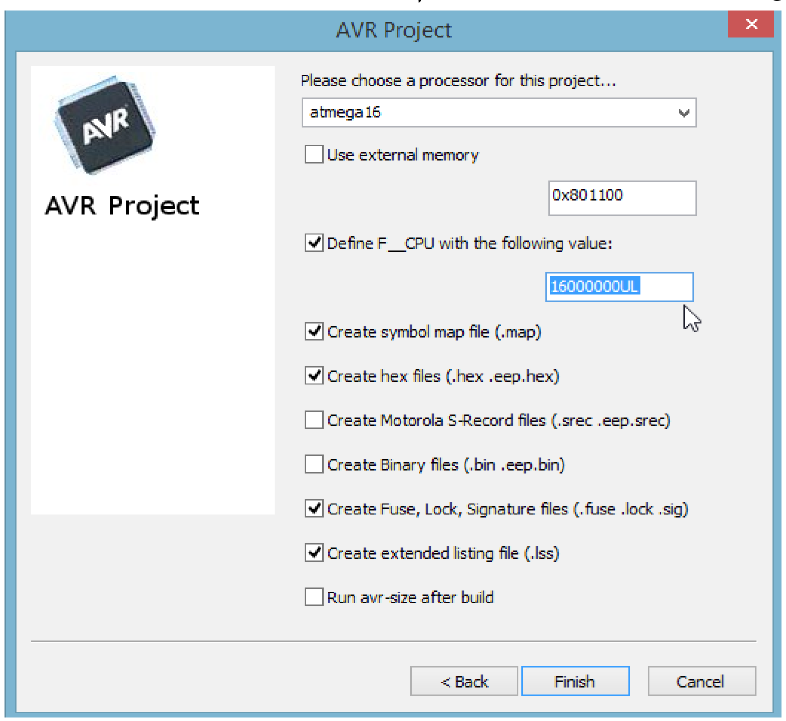


قم بكتابة اسم المشروع ATmega16_blinking_led واختر المكان الذي تريد أن تحفظ به ملفات المشروع كما في الصورة التالية.

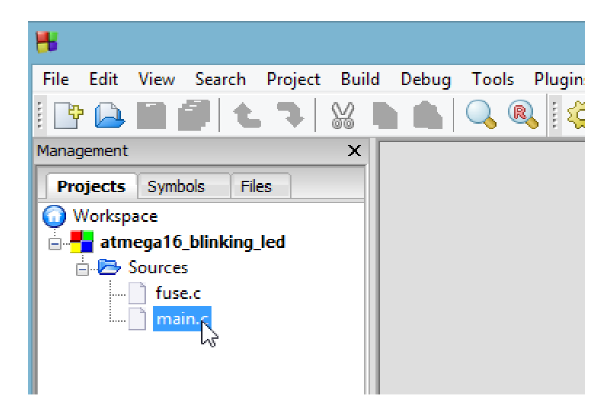


في الصفحة التالية قم بالتأشير على خيار Create Release فقط وألغ التأشير على Create "Debug" configuration

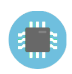
ستظهر صفحة جديدة خاصة بإعدادات المُتحكِم الدقيق، من القائمة العلوية اختر نوع المُتحكِم ATmega16 ثم قم بتغير سرعة تشغيل المعالج إلى التردد المطلوب مثل 8 ميجا هرتز بدل من 16 ميجا وذلك عبر كتابة الرقم 8000000



الآن أصبح كل شيء جاهز لنكتب أول برنامج،حيث سنجد أن برنامج CodeBlocks قام بعمل ملفين الأول يسمى main.c و الثاني هو fuses.c - سنتعامل مع الملف الأول فقط وهو الملف الذي سنكتب فيه البرامج الخاصة بالتحكم، أما الفيوزات فيمكنك أن تقوم باعدادها مباشرة كما هو مذكور في فصل الفيوزات.



للبدء قم بالضغط مرتين على ملف main.c من القائمة الجانبية للبرنامج لتظهر الصفحة



الخاصة بكتابة الكود البرمجي وبعد الانتهاء من كتابة الكود اضغط على زر Build من الشريط العلوي للبرنامج (الزر الذي يمتلك رمز الترس) كما في الصورة التالية:

main.c [atmega16_blinking_led] - Code::Blocks 13.12

ld Debug Tools Plugins Settings Help

Debug

main.c

Build

```c
/*
Hello AVR World !
Make a Blinking led with atmega16 and led on PORTC(pin PC0)
 */

#include <avr/io.h>
#include <avr/delay.h>

int main(void)
{

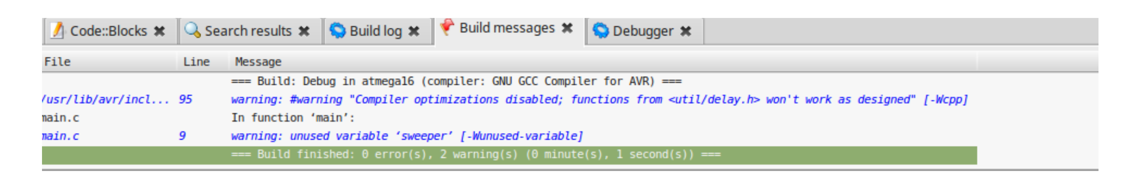
    DDRC = 0b11111111;

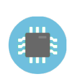
    while(1)
    {
        PORTC = 0b00000000;
        _delay_ms(500);

        PORTC = 0b00000001;
        _delay_ms(500);
    }

    return 0;
}
```

ستظهر في القائمة السفلية للبرنامج عبارة تدل على نجاح عملية التجميع Compilation بدون أخطاء errors . الآن يصبح لدينا ملف الـ hex الذي يمكننا استخدامه إما لبرمجة المُتحكِم أو يمكننا استخدامه مع برنامج المحاكاة بروتس Protues لعمل محاكاة للمُتحكِم ATmega16





مُلحق: ترجمة الملفات باستخدام makefile

تُعد هذه الطريقة هي أسرع اسلوب لترجمة الملفات المكتوبة بلغة السي وتحويلها إلى الصيغ التنفيذية مثل ملفات hex (أو حتى ترجمة البرامج التقليدية الخاصة بالحاسب الآلي).

تعتمد هذه الطريقة على استخدام المترجم avr-gcc مباشرة دون الحاجة لوجود أي بيئة تطوير مثل Atmel studio أو Codeblocks. كما تمتاز بأنها **معيارية** وصالحة للعمل على جميع نظم التشغيل windows, linux, mac وبسبب عدم الحاجة لوجود IDE نجد أن هذه الطريقة هي الأخف على الإطلاق لذا لا تستعجب إذا وجدت أن معظم المشاريع الموجودة على الإنترنت تستخدم هذه الطريقة في الترجمة.

شخصياً أستخدم هذ الأسلوب في جميع المشاريع التي أعمل عليها حيث استخدم محرر النصوص NotePad++ أو sublime لكتابة الملفات بلغة السي ثم أحولها إلى ملفات hex باستخدام الـ makefile (سواء على ويندوز أو لينكس).

التجربة الأولى

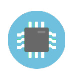
في البداية تأكد من وجود الـ toolchain على جهازك (على نظام ويندوز تسمى WinAVR toolchain) ويمكن تحميلها من الرابط التالي
http://sourceforge.net/projects/winavr/files

لاختبار تنصيب الـ toolchain بصورة صحيحة، قم بفتح **سطر الأوامر** واكتب الأمر make

- سطر الأوامر على نظام ويندوز يسمى command prompt ويتم تشغيله بكتابة بالأمر cmd من قائمة start
- سطر الأوامر على نظام لينكس يسمى terminal (الطرفية)

إذا كانت الـ toolchain منصبة بصورة صحيحة يفترض أن تظهر الرسالة التالية

```
make: *** No targets specified and no makefile found.  Stop.
```



```
Command Prompt

c:\>make
make: *** No targets specified and no makefile found.  Stop.

c:\>
```

والآن لنقم بتجربة الملف الأول وهو المثال blinking led، قم بفتح المجلد المسمى compile with make file لتجد بداخله ملفين وهما

```
main.c
makefile
```

الملف main.c يحتوي على المثال الأول في الكتاب blinking led أما الملف makefile فيحتوي على جميع الإعدادات الخاصة بتحويل الملف main.c إلى ملف الهيكس. قم بفتح الملف باستخدام أي محرر نصوص مثل notepad++

```
C:\Users\Abdallah\Documents\Blinking Led Example\makefile - Notepad++
File  Edit  Search  View  Encoding  Language  Settings  Macro  Run  Plugins  Window  ?
makefile

#####        Target Specific Details          #####
#####     Customize these for your project    #####

# Name of target controller
# (e.g. 'at90s8515', see the available avr-gcc mmcu
# options for possible values)
MCU=atmega16

# id to use with programmer
# default: PROGRAMMER_MCU=$(MCU)
# In case the programer used, e.g avrdude, doesn't
# accept the same MCU name as avr-gcc (for example
# for ATmega8s, avr-gcc expects 'atmega16' and
# avrdude requires 'm16')

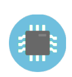
PROGRAMMER_MCU=m16

# Name of our project
# (use a single word, e.g. 'myproject')
PROJECTNAME=blinkingled

# Source files
# List C/C++/Assembly source files:
# (list all files to compile, e.g. 'a.c b.cpp as.S'):
# Use .cc, .cpp or .C suffix for C++ files, use .S
# (NOT .s !!!) for assembly source code files.
PRJSRC=main.c
```

يمتلك هذا الملف الكثير من الإعدادات وأهمها هي المجموعة التالية:

```
MCU=ATmega16
```



هنا يتم وضع اسم المُتحكِم الدقيق المستخدم والذي سيتم توليد ملف الهيكس خصيصاً له (مع ملاحظة أن ملفات الهيكس تختلف من مُتحكِم لأخر).

```
PROGRAMMER_MCU=m16
```

هذا الخيار يحدد اسم المُتحكِم الذي سيرفع عليه ملف الهيكس باستخدام برنامج avrdude ويجب تسميته بالحرف m ثم رقم المُتحكِم فمثلا

- ATmega16 = m16
- ATmega32 = m32
- atmega328p = m328p

```
PROJECTNAME=blinkingled
```

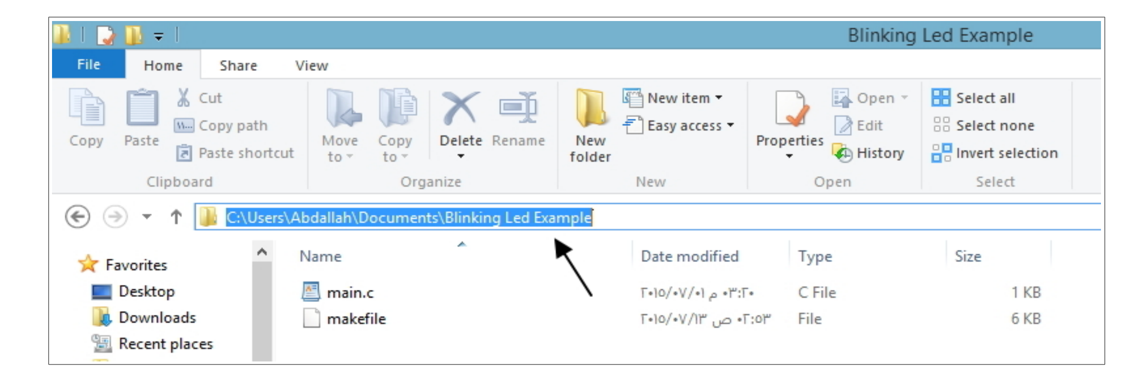
هذا هو اسم المشروع وسيكون اسم ملف الهيكس الذي سيتم توليده

```
PRJSRC=main.c
```

اكتب هنا اسم الملف الذي يحتوي على الكود البرمجي المراد تحويله إلى ملف هيكس.

احفظ الملف ثم توجه إلى سطر الأوامر وحول مسار المجلد الحالي إلى نفس مكان الملفين main.c و makefile فمثلاً- المسار الذي استخدمه هو

```
C:\Users\Abdallah\Documents\Blinking Led Example
```



قم بكتابة الأمر "cd "path واستبدل كلمة path بالمسار ثم اضغط Enter لتجد أن سطر الأوامر انتقل إلى المجلد المطلوب كما في الصورة التالية

Command Prompt

```
c:\>cd "C:\Users\Abdallah\Documents\Blinking Led Example"

C:\Users\Abdallah\Documents\Blinking Led Example>_
```

والآن اكتب الأمر make hex لتجد أن المترجم بدأ عملية تحويل الملف main.c إلى الملف blinkingled.hex كما في الصورة التالية

Command Prompt - make hex

```
C:\Users\Abdallah\Documents\Blinking Led Example>make hex
makefile:205: warning: overriding commands for target `.c.o'
makefile:200: warning: ignoring old commands for target `.c.o'
avr-gcc -I. -I/path/to/include -g -mmcu=atmega16 -Os -fpack-struct -fshort-enums
 -funsigned-bitfields -funsigned-char -Wall -Wstrict-prototypes -Wa,-ahlms=main.
lst -fno-exceptions -Wa,-ahlms= -c main.c -o main.o
In file included from main.c:9:
c:/winavr-20100110/lib/gcc/../../avr/include/avr/delay.h:36:2: warning: #warning
 "This file has been moved to <util/delay.h>."
```

شكل المجلد بعد أن تم توليد الملف blinkingled.hex

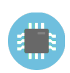
C ▸ Documents ▸ Blinking Led Example

Name	Date modified	Type	Size
blinkingled.ee.hex	٢٠١٥/٠٧/١٣ ص ٠٣:١١	HEX File	1 KB
blinkingled.hex	٢٠١٥/٠٧/١٣ ص ٠٣:١١	HEX File	1 KB
blinkingled.out	٢٠١٥/٠٧/١٣ ص ٠٣:١١	OUT File	4 KB
blinkingled.out.map	٢٠١٥/٠٧/١٣ ص ٠٣:١١	Linker Address Map	10 KB
main.c	٢٠١٥/٠٧/٠١ م ٠٣:٢٠	C File	1 KB
main.o	٢٠١٥/٠٧/١٣ ص ٠٣:١١	O File	4 KB
makefile	٢٠١٥/٠٧/١٣ ص ٠٢:٥٣	File	6 KB

مسح الملفات من خلال **make**

بافتراض أنك تريد مسح جميع الملفات التي نتجت من عملية الترجمة فيمكنك ذلك بسهولة باستخدام الأمر make clean

ملاحظة:ملف makefile المرفق مع الكتاب هو نسخة مبسطة من هذا النوع من الملفات وقد تجد في مشاريع أخرى نفس الملف ولكن بإعدادات أكثر كما سنرى في الفصل الخاص بـ Real Time Operating system
يستحسن أن تعتاد وتتقن ترجمة الملفات بهذه الطريقة لأنها تعتبر الطريقة المعتمدة في مجتمعات المطورين وستجد الكثير من المشاريع على الإنترنت تستخدم هذه الطريقة.



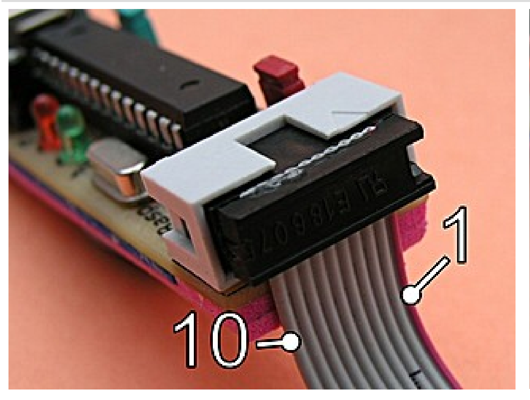
مُلحق: رفع ملف الـ Hex على المُتحكِم الدقيق

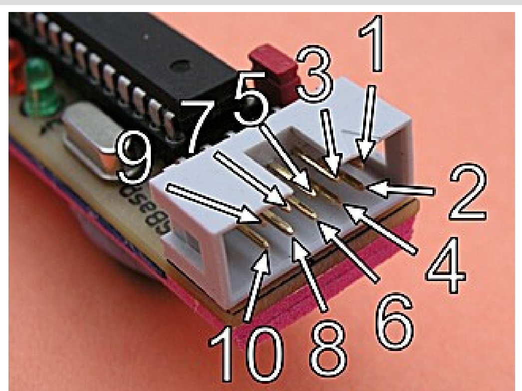
كيف توصل المبرمجة بالمتحكم

تمتلك أغلب مبرمجات الـAVR إما 6 أو 10 أطراف(تسمى هذه الأطراف ISP connector) وتكون مرتبه كما هو موضح بالصورة التالية

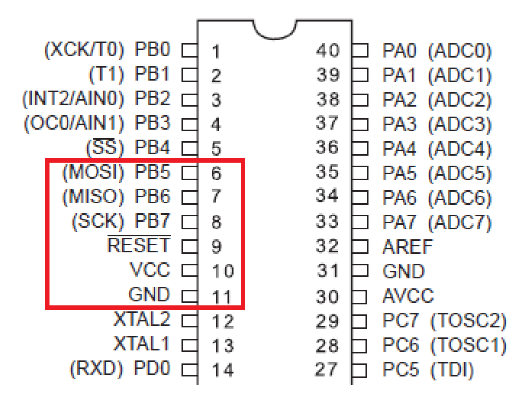


مثال على ذلك المبرمجة USBasp

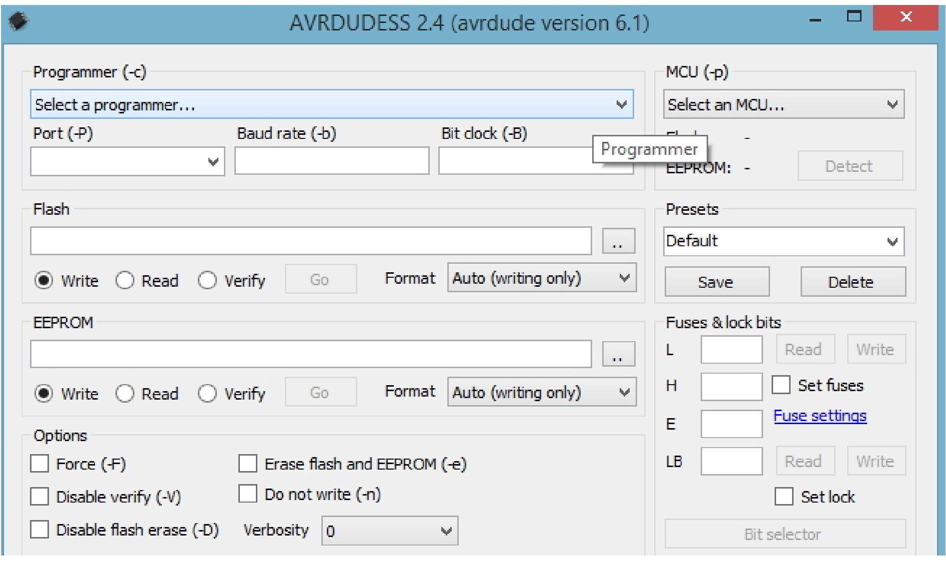




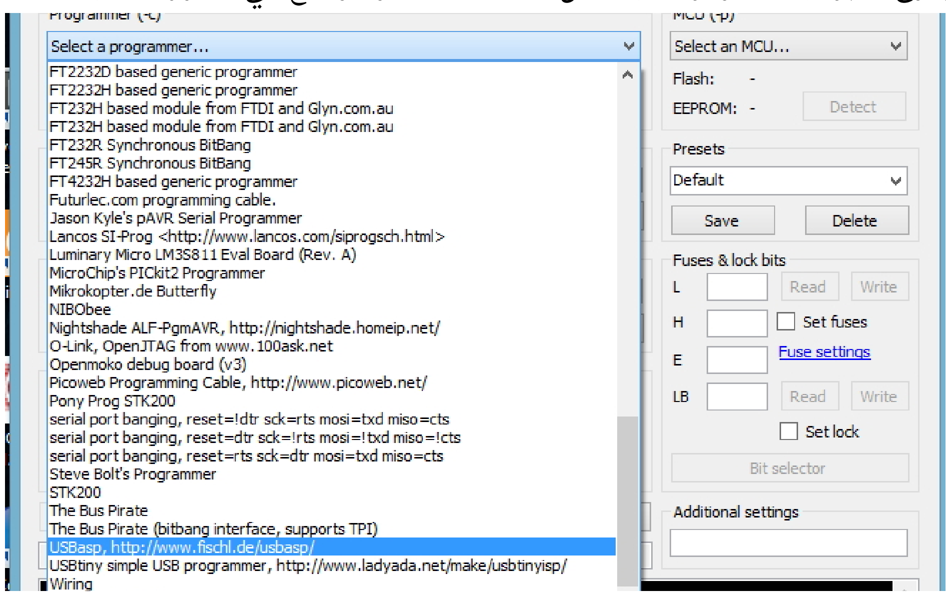
لبرمجة المُتحكِم الدقيق كل ما عليك فعله هو توصيل كل طرف في المبرمجة بما يوازية في المُتحكِم الدقيق وهذا يشمل الأطراف MOSI – MISO – SCK – RST(RESET) والطرف 5V في المبرمجة يتصل بال VCC مع AVCC وكذلك يتم توصيل كل اطراف الـ GND ببعضها (المُتحكِم والمبرمجة).



بعد الانتهاء من توصيل دائرة المُتحكِم على لوحة التجارب Breadboard سنقوم برفع ملف الهيكس باستخدام برنامج avrdude وذلك من خلال الواجهة الرسومية AVRdudess. في البداية قم بفتح البرنامج AVRdudess

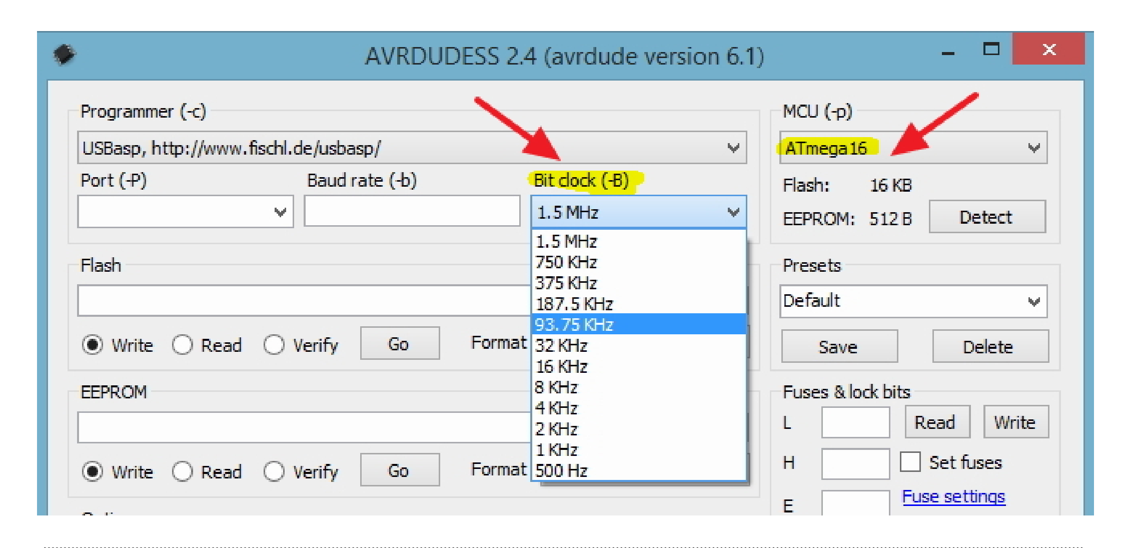


اختر نوع المُبرمجة المتوفرة لديك مثل USBasp كما هو موضح في الصورة التالية:



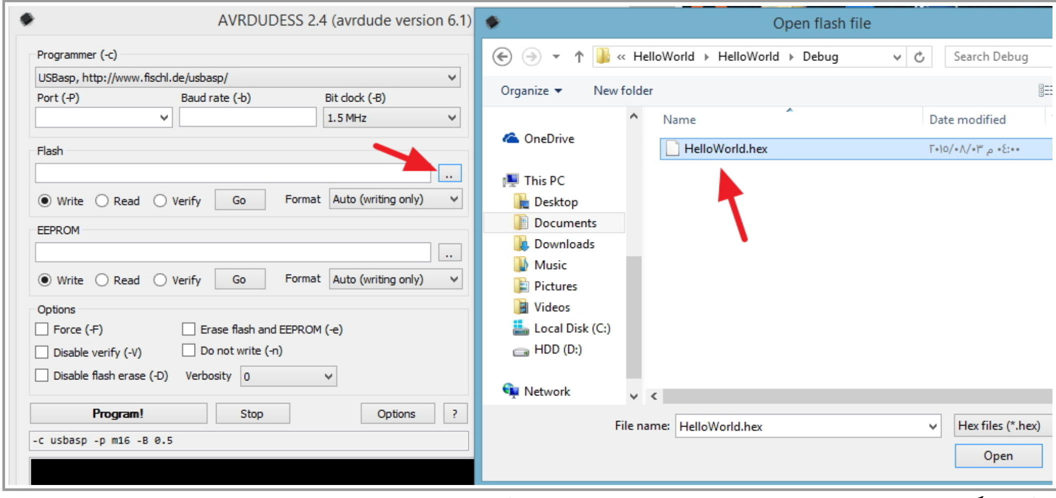
بعد اختيار المُبرمجة قم باختيار نوع المُتحكِم الدقيق (من القائمة الموجود على الجانب الأيمن من البرنامج) وفي حالة أن المُتحكِم يعمل بدائرة المذبذب الداخلي بسرعة **1 ميجا** يجب أن تغير سرعة رفع البرنامج لتصبح **93 كيلوبايت في الثانية** أما إذا كان المُتحكِم يعمل

باستخدام دائرة مذبذب خارجي مثل الكريستالة 16 ميجا فيمكنك أن تترك خيار سرعة الرفع يساوي الخامس ميجا كما هو موضح في الصورة التالية.

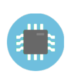


ملاحظة: يستطيع برنامج AVRdudess أن يتعرف على المُتحكِم بصورة تلقائية وذلك عبر الضغط على زر Detect الموجود على جانب الشاشة الأيسر.

والآن اختر ملف الهيكس الذي يحتوي على البرنامج المطلوب رفعة إلى المُتحكِم الدقيق



وأخيراً قم بالضغط على زر **Program** الموجود بأسفل شاشة البرنامج



استخدام برنامج avrdude بدون واجهة رسومية

بالرغم من وجود العديد من الواجهات الرسومية لبرنامج avrdude إلا أنه بالأساس يعمل من خلال سطر الأوامر (سواء cmd على ويندوز أو الـ shell على نظام لينكس).

قد يبدو استخدام avrdude مزعجاً أو مخيفاً لمن لا يُستخدَم نظام لينكس و ليس على احتكاك بسطر الأوامر إلا أنه سنكتشف سوية مدى سهولة استخدامه.

الشكل العام لتعليمة avrdude

```
avrdude -c avrisp -b 19200 -p atmega8 -P /dev/ttyACM0 -U flash:w:main.hex
```

avrdude: اسم البرنامج

avrisp: اسم المبرمجة
arduino
إذا أردت أن تبرمج الأردوينو من خلاله (بأية حال برنامج الأردوينو يعتمد على avrdude في عملية الرفع)
avrisp
usbasp

19200: سرعة الاتصال baudrate عبر المنفذ التسلسلي

atmega8: المتحكم المراد برمجته

/dev/ttyACM0: اسم المنفذ التسلسلي المستخدم يختلف حسب نظام التشغيل

flash:w:main.hex: الأمر المراد تنفيذه
memtype:op:filename

flash , eeprom , fuse , efuse , lfuse , hfuse , lock ...

r:read
w:write
v:verify

اسم الملف المراد استخدامه في العملية المحددة ، و يجب أن يكون مسار سطر الأوامر في نفس مسار وجود هذا الملف

atadiat.com
عتاديات

أهم الخيارات المتاحة

اسم المبرمجة : بكتابة c- كخيار في سطر الأمر avrdude لتحديد نوع المبرمجة مثلاً : arduino – avrisp – usbasp –.. إلخ.

المُتحكِم المراد برمجته : بكتابة p- كخيار في سطر الأمر avrdude لتحديد اسم المُتحكِم.

سرعة الاتصال : بكتابة b- كخيار في سطر الأمر ، و هذا الخيار مهم لتأكيد كون سرعة الاستقبال في الكود المنفذ على مُتحكِم " المُبرمجة" يساوي سرعة avrdude لأنه اختلاف السرعتين سيؤدي إلى أخطاء في التزامن.

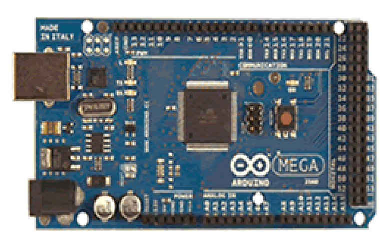
المنفذ التسلسلي : بكتابة P- كخيار في سطر الأمر . اسم المنفذ يتخلف من نظام تشغيل لآخر فغالباً ما يكون من نمط **dev/ttyACM** بالنسبة للينكس و **COM** بالنسبة للويندوز .

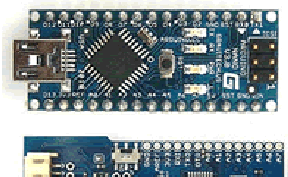
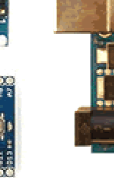
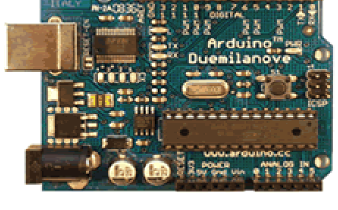
الأمر التنفيذي : بكتابة U- كخيار في سطر الأمر ، و من ثم تحديد الأمر بالصياغة التالية : **memtype:op:filename:format** و يقصد بـ format نوع الملف المستخدم حيث يمكن استخدام الملفات من نوع hex أو bin.أما filename فيتم استبدالها بأسم الملف، وكلمة op تعني operation وهي العملية المطلوب تنفيذها مثل read (اقرأ الذاكرة) أو write (أكتب في الذاكرة) أو verify والتي تعني التأكد من أن محتوى الذاكرة يطابق ملف الهيكس الذي يتم تحديد أسمه في نفس الأمر.

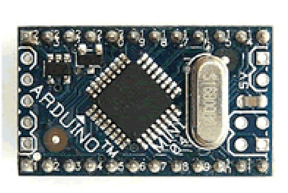
خيارات مختلفة : من الظلم إختصار خيارات avrdude بهذه الأسطر القليلة و لكل خيار حالة استخدام خاصة و هي مشروحة بوضوح في كتيب البرنامج و يمكن الاطلاع عليها من خلال كتابة man avrdude على سطر الأوامر في لينكس .

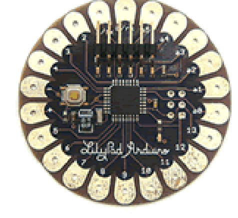
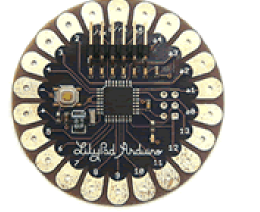
مُلحق: كيف تستخدم لوحات آردوينو لتعلم برمجة AVR

تعد لوحات آردوينو Arduino من أرخص اللوحات التطويرية في العالم حيث تبدأ أسعارها من 5 دولار فحسب مثل لوحة Arduino nano. وما يميز هذه اللوحات أنها تعمل بمتحكمات AVR وبالتحديد عائلة Atmega (هناك لوحات آردوينو تعمل بعائلة ATTiny أيضاً).





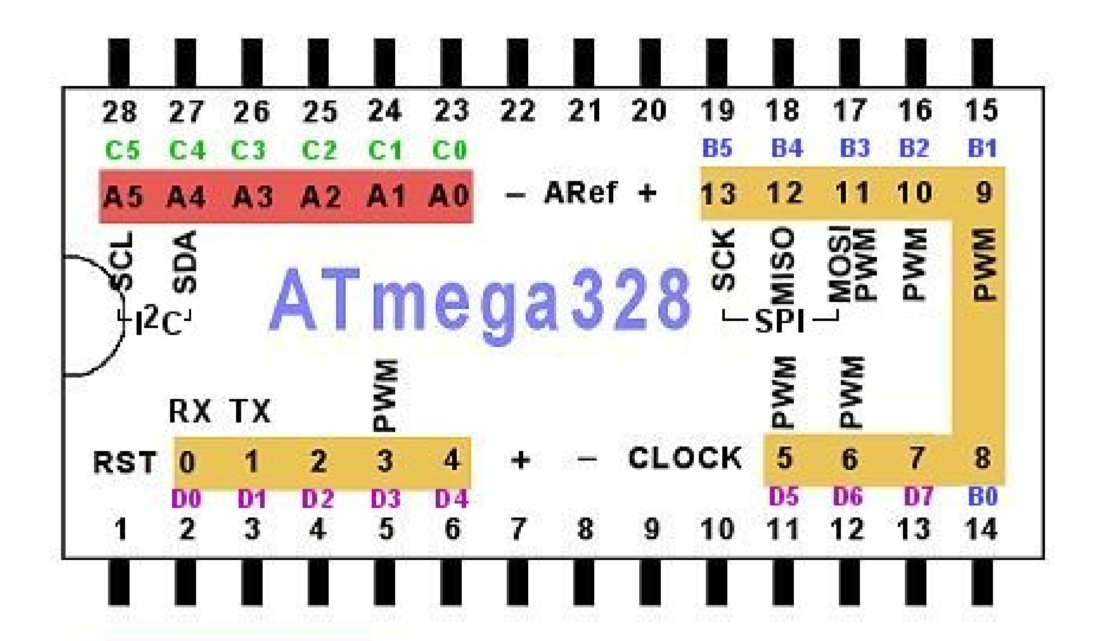





الحقيقة أن برنامج Arduino IDE ماهو إلا المترجم الشهير AVR-GCC الذي نستخدمه في هذا الكتاب+الواجهة الرسومية الخاصة ببرنامج Processing ومضاف إليه العديد من المكتبات البرمجية من مشروع Wiring + بعض التعديلات البسيطة على برنامج avrdude. وهذا يجعل برنامج آردوينو متوافق تماماً مع البرامج المكتوبة بلغة الـ ANSI – C.

هناك أمر واحد يجب الانتباه له عند التعامل مع لوحات آردوينو وهو **"ترقيم الأطراف"**، حيث نجد أن مصممي لوحات آردوينو لديهم أسلوب مختلف لترقيم أطراف مُتحكِمات Atmega الموجودة على اللوحات ولا يتم استخدام أسماء البورتات مثل port A, port B وإنما يتم استخدام ترقيم بسيط مثل 0,1,2,3

على أي حال هذا الأمر لا يمثل مشكلة فكل ما عليك معرفته هو أسماء الأطراف عند برمجتها. لنأخذ لوحة آردوينو Uno كمثال (باعتبارها أشهر لوحات آردوينو). الصورة التالية تمثل ترقيم أطراف المُتحكِم ATmega328 بكل من الأسلوب الأصلي (مثل ما هو مذكور في دليل البيانات + ترقيم آردوينو).



Digital Input/Output
Analog / Digital

كما نرى في الصورة السابقة نجد أن الترقيم المكتوب في المربعات البرتقالية والحمراء هو ترقم آردوينو بينما الترقيم المكتوب بالحروف الزرقاء هو الترقيم الأصلي للأطراف ويعبر عن اسم البورت مثل D0 تعني port D pin 0.

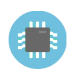
والآن لنقم بكتابة برنامج Blinking Led على طريقة الـ ANSI – C

> ملاحظة: تحتوي معظم لوحات آردوينو على دايود ضوئي متصل بالطرف رقم 13 وهو في الحقيقة الطرف PB5 (البورت B – الطرف الخامس).

قم بفتح برنامج Arduino IDE وأكتب برنامج الـ Blinking led كما هو مشروح في المثال الأول في الفصل الثالث من الكتاب. مع تغيير بسيط وهو تعريف سرعة المعالج برقم 16 ميجا

```
#define F_CPU 16000000UL
```

بعد الانتهاء من كتاب البرنامج قم بالضغط على زر Upload لتجد أن برنامج Arduino قام



بترجمة الملف وتحويله إلى Hex file كما هو موضح بالصورة التالية

sketch_aug23a | Arduino 1.6.5

File Edit Sketch Tools Help

sketch_aug23a

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/delay.h>

int main(void)
{

    DDRB |= (1<<PB5);

    while(1)
    {
        PORTB |= (1<<PB5);
        _delay_ms(500);

        PORTB &= ~(1<<PB5);
        _delay_ms(500);
    }
    return 0;
}

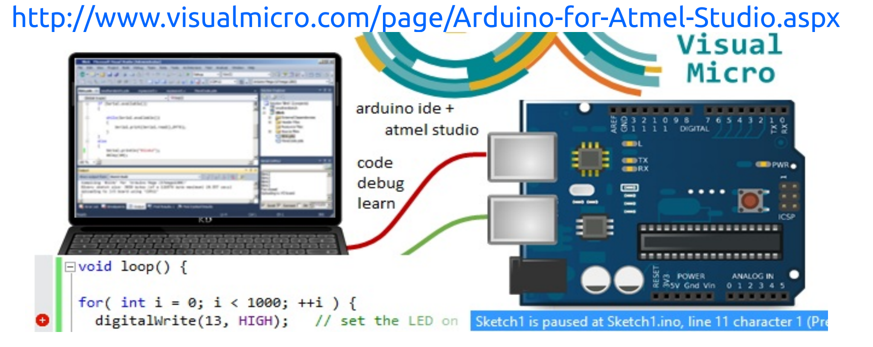
```

Done compiling.

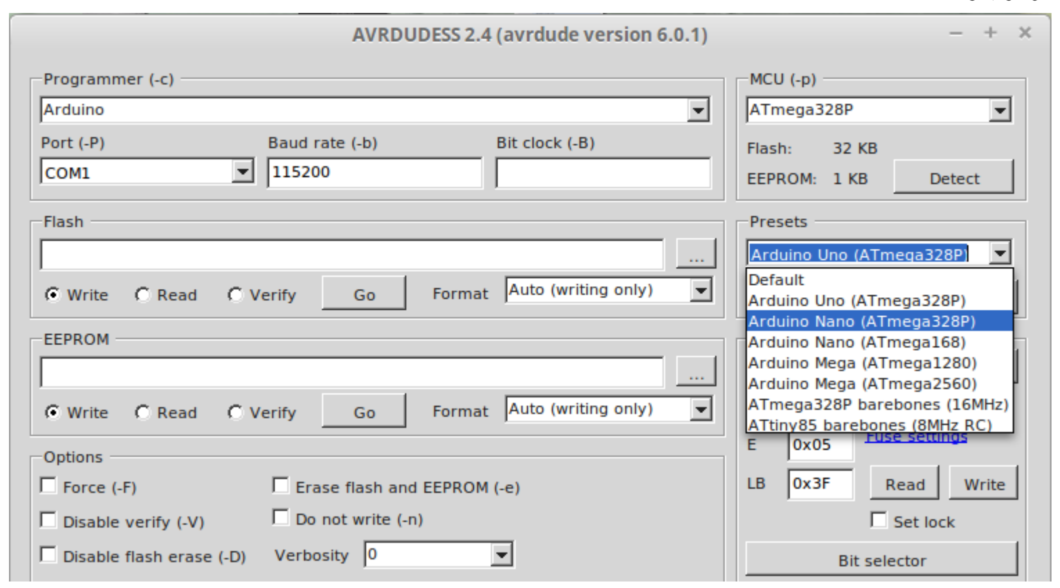
```
Sketch uses 176 bytes (0%) of program storage space. Maximum is
30,720 bytes.
Global variables use 0 bytes (0%) of dynamic memory, leaving 2,048
bytes for local variables. Maximum is 2,048 bytes.
```

إذا لم تكن تفضل استخدام برنامج آردوينو في كتابة الكود فيمكنك أن تستخدم برنامج Atmel Studio مع إضافة الدعم المباشر للوحات آردوينو بسهولة من خلال إضافة visualmicro والتي تجعل برنامج Atmel studio يتعامل مع لوحات آردوينو مباشرة

http://www.visualmicro.com/page/Arduino-for-Atmel-Studio.aspx



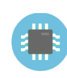
أيضاً يمكنك استخدام ملفات makefile في ترجمة الأكواد البرمجية وتحويلها إلى ملف هيكس (راجع ملحق شرح makefile) ثم استخدام برنامج ARVdudess لرفعها على لوحات آردوينو



أيضاً بامكانك استخدام لوحات آردوينو كمبرمجة Programmer لأي مُتحكِم AVR وذلك عبر برنامج ArduinoISP + AVRdudess

ملاحظة: بامكانك أيضاً كتابة برامج بلغة الأسمبلي داخل برنامج
Arduino IDE و التجربة التالية تشرح برنامج Blinking Led بلغة الأسمبلي

https://ucexperiment.wordpress.com/2013/05/31/arduino-blink-using-gcc-inline-/assembly



قائمة المراجع



دورة الإلكترونيات العملية د. وليد عيسى (أساسيات الإلكترونيات من الصفر)
https://www.youtube.com/playlist?list=PLww54WQ2wa5rOJ7FcXxi-CMNgmpybv7ei

----

دورة المهندس وليد بليد في شرح برمجة مُتحكِمات AVR بإستخدام لغة Bascom
https://www.youtube.com/playlist?list=PLww54WQ2wa5qWSTU7MYqjN0jHNaWuoXUk

----

قناة شركة ENG Unity وتحتوي على العديد من الدورات العربية عن النظم المدمجة ويتضمن ذلك AVR و برنامج Altium والتصميم الرقمي Digital Circuits

https://www.youtube.com/user/ENGUnity/playlists

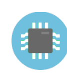
قناة عربية تحتوي على دورات فيديو مبسطة عن النظم المدمجة وتتضمن
C for Embedded System
Micrcontroller Architectre

https://www.youtube.com/channel/UCbZ7PLd5LAnje1hpyoiRW0A/playlists

----

دورة تعلم برمجة AVR بإستخدام برنامج CodeVision
http://www.qariya.info/vb/showthread.php?t=81782

موقع "عتاداديات" يحتوي على مجموعة من المقالات المبسطة
http://www.atadiat.com/%D8%A7%D9%84%D9%82%D8%B3%D9%85-%D8%A7%D9%84%D8%AA%D8%B9%D9%84%D9%8A%D9%85%D9%8A/



مراجع تعليمية إنجليزية

مرجع AVR العملاق (يعد من أفضل المراجع في العالم للنظم المدمجة)

AVR Microcontroller and Embedded Systems: Using Assembly and C (Pearson Custom Electronics Technology) - Muhammad Ali Mazidi

http://www.amazon.com/AVR-Microcontroller-Embedded-Systems-Electronics/dp/0138003319

من أفضل المراجع التي تعلمك "كيفية تحويل متطلبات العميل إلى أفضل تصميم برنامج على النظم المدمجة"

An Embedded Software Primer

http://www.amazon.com/Embedded-Software-Primer-David-Simon/dp/020161569X

دورة **NewBieHack** لتعلم AVR في 55 درس مفصل، وتعتبر من أفضل الدورات المبسطة والممتعة

https://www.youtube.com/playlist?list=PLE72E4CFE73BD1DE1

المرجع الرسمي لنظام **FreeRTOS**

Using the FreeRTOS Real Time Kernel - Standard Edition

http://www.amazon.com/Using-FreeRTOS-Real-Time-Kernel/dp/1446169146/
